N° d'ordre: 50

THÈSE

présentée à

L'UNIVERSITÉ DE BRETAGNE SUD

EN HABILITATION CONJOINTE AVEC L'ÉCOLE NATIONALE SUPÉRIEURE DE TÉLÉCOMMUNICATIONS DE BRETAGNE

pour obtenir le titre de

DOCTEUR DE l'UBS

Mention : *Sciences pour l'ingénieur*

par

David GNAEDIG

High-Speed decoding of convolutional Turbo Codes

Décodage haut débit de turbocodes convolutifs

Soutenue le 24 juin 2005 devant le jury composé de :

Pierre Duhamel	Président et rapporteur
Christian Schlegel	Rapporteur
Guido Montorsi	Examinateurs
Claude Berrou	
Jacky Tousch	
Emmanuel Boutillon	Directeur de thèse
Michel Jézéquel	Co-directeur de thèse

Remerciements

Je tiens à exprimer ici toute ma reconnaissance aux personnes qui m'ont aidé, encouragé, soutenu ou tout simplement supporté tout au long de mon travail de thèse.

Je souhaite tout d'abord remercier Pierre Duhamel de Supélec pour m'avoir fait l'honneur de présider mon jury de thèse ainsi que pour son rôle de rapporteur. Mes plus vifs remerciements s'adressent au professeur Christian Schlegel de l'université d'Alberta pour avoir lui aussi cordialement accepté d'être rapporteur de ce travail. Ses commentaires m'ont permis d'ouvrir de nouvelles perspectives à mon travail. Je tiens également à remercier Guido Montorsi, professeur au Politechnico de Turin, pour avoir accepté de faire partie de mon jury de thèse et pour son intérêt pour mon travail. J'ai apprécié les échanges fort intéressants que nous avons pu avoir concernant les architectures de décodage. Ses commentaires élogieux tout comme ceux de Pierre et Christian m'ont touché. Qu'ils trouvent ici l'expression de ma sincère reconnaissance.

Je veux également remercier Claude Berrou, professeur à l'ENST de Bretagne pour m'avoir non seulement fait le plaisir d'évaluer mes travaux, mais aussi pour sa disponibilité et les nombreuses discussions au sujet des turbocodes que nous avons pu avoir. Son esprit visionnaire et son expérience m'ont été fort utiles.

Je tiens à exprimer tout ma gratitude et mon amitié à Emmanuel Boutillon, mon directeur de thèse, qui m'a permis de développer mon goût pour la recherche et qui m'a guidé durant ces 3 années. Par sa disponibilité de tous les instants, son investissement personnel ainsi que par son enthousiasme et sa persévérance, il m'a permis de faire ressortir le meilleur de moi-même tout au long de ces années. Les résultats que nous avons pu obtenir ensemble en témoignent. Emmanuel, je ne saurais trop te remercier pour tout ce que tu m'a apporté dans le cadre et en dehors de ce travail.

Je remercie également Michel Jézéquel, co-directeur de thèse et directeur du laboratoire d'Electronique de l'ENST Bretagne pour m'y avoir accueilli et pour la qualité de son encadrement. Sa disponibilité, ses remarques judicieuses et ses qualités de synthèse sont autant de raisons pour lesquelles sont encadrement fût extrêmement profitable.

Je n'oublie évidemment pas TurboConcept, Nathalie Brengarth et Jacky Tousch en particulier, qui m'ont accueilli et intégré au sein de la société. Le risque qu'ils ont pris en finançant ces recherches les honorent. La confiance qu'ils m'ont accordée ainsi que la liberté qu'ils m'ont laissée tout au long de ces trois années favorisa grandement mon épanouissement scientifique et professionnel. Un grand merci à Jacky, qui a pris le temps de suivre avec mes travaux et avec intérêt et avec qui ce fût et ça reste un plaisir de travailler. La qualité de ce rapport s'est vue remarquablement améliorée par sa relecture attentive et pointilleuse, et ses nombreuses corrections. Un grand merci également à mes collègues de TurboConcept avec qui j'ai pris plaisir à travailler dans un cadre agréable. Merci à Nghia pour avoir éclairé mon esprit sur les aspects marketing et commercial de notre travail. Merci enfin à Big qui excelle

dans la construction de tunnels et l’élevage de pingouins sans qui l’informatique serait une jungle incontrôlable.

Je tiens aussi à remercier Janet Ormrod, professeur d’anglais à l’ENST Bretagne, pour sa gentillesse et le temps qu’elle a consacré à corriger mes articles. Je lui dois mes progrès importants en anglais ainsi que l’amélioration considérable de la qualité de mon rapport.

Merci également à mes collègues du LESTER, où Eric Martin a eu la gentillesse de m’accueillir, et du département Electronique de l’ENST Bretagne. En premier lieu à mes collègues doctorants avec que j’ai pris plaisir à connaître: Philippe pour son intérêt pour les entrelaceurs, Amor pour son amitié, Matthieu pour son intérêt pour les roulettes, Slim, Frédéric et Sylvain pour leur visite en Bretagne, et tous les autres que je ne peux citer ici. Merci aussi à tous les permanents qui ont contribué à développer un environnement de travail agréable, qui m’ont apportés des conseils précieux ou tout simplement un soutien logistique. Je pense également à Vincent Gaudet de passage à l’ENST Bretagne, qui a été à l’origine de mes travaux.

Merci aussi à tous mes amis danseurs et danseuses de salsa, de danses irlandaises, de fest-noz, à mes amis de Brest, de Lorient et Paris, grâce à qui ces années m’ont paru bien trop courtes.

Merci enfin à ma famille et à Valérie pour m’avoir accompagné et soutenu tout au long de cette aventure.

Résumé

Les turbocodes sont des codes obtenus par une concaténation de plusieurs codes convolutifs séparés par des entrelaceurs. En 1993, ils ont révolutionné le domaine du codage correcteur d'erreurs en s'approchant à quelques dixièmes de décibels de la limite théorique de Shannon. Ces performances sont d'autant plus remarquables que le principe itératif permet d'en effectuer le décodage avec une complexité matérielle limitée. Le succès des turbocodes s'est traduit par leur introduction dans plusieurs standards de communication. Les besoins croissants dans le domaine des réseaux large bande, nécessitent des implantations hauts débits qui posent de nouvelles problématiques.

L'objectif de cette thèse est d'étudier des architectures de décodage à haut débit offrant le meilleur compromis en terme de débit sur complexité. Dans un premier temps, nous avons proposé un modèle simple permettant d'exprimer le débit et l'efficacité d'une architecture. Ce modèle appliqué au turbo-décodage met en évidence trois paramètres caractéristiques ayant un impact sur le débit et l'efficacité du décodeur : le degré de parallélisme, le taux d'utilisation (activité) des unités de calcul et la fréquence d'horloge. Nous avons abordé chacun de ces points en explorant un large spectre de possibilités de l'espace de conception allant de la construction conjointe du code et du décodeur à l'optimisation directe des architectures de décodage pour un code ou un ensemble de codes prédéfinis.

Nous avons tout d'abord proposé un nouveau schéma de codage appelé *turbocodes à roulettes* permettant de minimiser la mémoire du décodeur par un décodage en parallèle d'un mot de code reçu par plusieurs processeurs à entrée et sortie souples. Afin de résoudre le problème des accès concurrents aux mémoires qui en résulte, nous avons conçu un nouvel entrelaceur hiérarchique. Nous avons ensuite exploré plusieurs solutions permettant d'améliorer l'activité des processeurs : utilisation d'une architecture hybride série/parallèle et proposition de nouveaux séquencements au niveau interne des processeurs, et aussi au niveau global en association avec la construction d'entrelaceurs contraints adaptés. Enfin, grâce à une méthode originale de réduction du chemin critique du calcul récursif des métriques de nœuds, nous avons obtenu, sans coût matériel supplémentaire pour un circuit FPGA, un doublement de la fréquence d'horloge du décodeur.

La plupart des techniques développées dans cette thèse ont été validées par la réalisation d'un turbo-décodeur pour le standard d'accès sans-fil large bande WiMAX (IEEE 802.16) qui atteint des performances de correction d'erreur excellentes pour un débit atteignant 100 Mbit/s sur un seul circuit FPGA.

Abstract

Turbo codes are built as a concatenation of several convolutional codes separated by interleavers. In 1993, they have revolutionized error correcting coding by approaching within a few tenths of a decibel the Shannon limit. This performance is even more astonishing because the iterative decoding principle enables the decoder to be implemented in hardware with a relative low complexity. Due to their success, they are now widely used in practical systems and open standards. The increasing demand for high throughput applications in broadband applications is strongly calling for high-speed decoder implementations, thus leading to new challenges.

The objective of this thesis is to study high-throughput decoding architectures offering the best throughput versus complexity trade-off. We first laid down a simple expression to evaluate the benefits of an architecture in terms of throughput and efficiency. The application of this model to turbo decoding highlighted three typical parameters influencing the throughput and efficiency of the decoder: the degree of parallelism, the ratio of utilization (activity) of the processing units and the clock frequency. We tackled each of these points by investigating a large spectrum of possibilities of the design space, ranging from the joint code and decoder design to the optimization of the decoder architecture for a given code or set of codes.

We first proposed a new coding scheme called Multiple Slice Turbo Codes making possible to minimize the memory requirements of the decoder using the parallel decoding of a the received codeword by several soft-input soft-output processors. In order to solve the resulting concurrent accesses to the memory, we designed a novel hierarchical interleaver. Second, we explored several solutions for improving the activity of the processors including the usage of a hybrid parallel/serial architecture and the introduction of two new schedules for parallel decoding: one schedule internal to the processors, and another at a more global level in association with an adapted constrained interleaver. Finally, thanks to an original method to reduce the critical path in the recursive computation of state metrics, we obtained, at no cost on a FPGA circuit, a doubling of the maximal clock frequency of the decoder.

Most of the new techniques developed in this thesis were validated by designing a turbo decoder for the wireless broadband access standard WiMAX (IEEE 802.16) that achieves excellent error decoding performance reaching a throughput of 100 Mbit/s on a single FPGA.

Preamble

The work presented in this thesis results from the collaboration of the three following entities:

- the LESTER laboratory (in French, Laboratoire d'Electronique des Systèmes TEmps Réel), Université de Bretagne Sud (UBS), Lorient, France.
- the TAMCIC laboratory (Traitement Algorithmique et Matériel de la Communication, de l'Information et de la Connaissance), Ecole Nationale Supérieure des Télécommunications de Bretagne, Brest, France.
- TurboConcept, an IP Core provider company in the domain of digital communications, Plouzané, France.

This thesis was funded by TurboConcept under the CIFRE (Convention Industrielle de Formation par la REcherche) convention N° 789/2001.

The collaboration started with the introduction of a new family of turbo codes that was devised for the efficient parallel decoding of turbo codes. This family of turbo code called Multiple Slice Turbo Codes (MSTCs) was patented in [Boutillon *et al.*-02] by UBS. The study of MSTCs and their applications is an important contribution of our work, leading us to tackle the concern of high-speed turbo decoding in a more general framework. In this framework, the optimization of high-speed turbo decoding implementation is conducted following several approaches in order to obtain the most efficient turbo decoder implementation, for a given throughput. This manuscript is presented following this general framework.

Préambule

Le travail présenté dans cette thèse est le fruit d'une collaboration des trois entités suivantes:

- le Laboratoire d'Electronique des Systèmes TEmps Réel (LESTER), FRE 2734 du CNRS, Université de Bretagne Sud (UBS), Lorient, France.
- le laboratoire TAMCIC (Traitement Algorithmique et Matériel de la Communication, de l'Information et de la Connaissance), UMR 2872 du CNRS, Ecole Nationale Supérieure des Télécommunications de Bretagne, Brest, France.
- TurboConcept, une société proposant des blocs de propriété intellectuelle (IP Cores) dans le domaine des communications numériques, Plouzané, France.

Cette thèse a été financée par TurboConcept dans le cadre de la convention CIFRE N° 789/2001.

Cette thèse a été financée par la société TurboConcept dans le cadre de la convention CIFRE (Convention Industrielle de Formation par la REcherche) N° 789/2001.

La collaboration entre les trois entités a débuté avec l'introduction d'une nouvelle classe de turbo codes construit dans le but d'implémenter efficacement une architecture de décodage parallèle. Cette famille the turbo code appelée Turbo Codes à Roulettes (TCRs) a été brevetée par l'UBS [Boutillon *et al.*-02]. L'étude des TCRs et de leur application est une contribution importante de notre travail, quit nous a ammené à considérer le probblem de turbo décodage à haut-débit suivant une cadre plus général. Dans ce cadre, l'optimization des implémentations de turbo décodage à haut débit s'effectue suivant plusieurs directions dans le but d'obtenir l'implémentation la plus efficace pour un débit visé. Ce manuscrit s'articule suivant ce cadre général.

Valorization of the thesis

Some key results and ideas provided in this thesis have been used successfully and included in commercial products by TurboConcept, providing key competitive advantages.

The author contributed to the active participation of TurboConcept in the DVB-S2 standardization committee. TurboConcept proposed a turbo code solution based on Multiple Slice Turbo Codes. In order to decrease the error floor, a serial concatenation with an outer Reed-Solomon code separated by a Forney interleaver was used.

The author was closely involved in the development of the turbo decoder family proposed by the company as commercial IP Core products addressing mainly broadband applications. In addition, the methodology developed to optimize the scaling factors has been used internally. Finally, the methodology for selecting interleaver parameters has led to commercial software.

Other key contributions of this thesis have not been used so far, but they also represent short term perspectives for future applications and products. They are currently under study.

Publications

Journal with review committee:

[1] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel, P. G. Gulak, "On Multiple Slice Turbo Codes", *Annales des Télécommunications*, vol. 60, no 1-2, Jan. 2005.

[2] D. Gnaedig, E. Boutillon, M. Jézéquel, "Design of Three-Dimensional Multiple Slice Turbo Codes", *EURASIP Journal on Applied Signal Processing, Special Issue on Turbo Processing*, vol 6., pp. 808-819, 2005.

[3] E. Boutillon, D. Gnaedig, "Maximum Spread of D-dimensional Multiple Turbo Codes", to be published in *IEEE Transactions on Communications*, 2005.

[4] D. Gnaedig, E. Boutillon, E. Martin, A. Nafkha, M. Jézéquel, J. Tousch, N. Brengarth. "High-level synthesis for behavioral design of the MAP algorithm for turbo decoder", in *Annales des Télécommunications*, vol. 59, no 3-4, pp. 325-348, 2004.

Conference papers:

[5] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel, P. G. Gulak, "On Multiple Slice Turbo Codes", in *Proceedings of the International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 343-346, Sept. 2003.

[6] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel, P. G. Gulak, "Les Turbo Codes à Roulettes", in *GRETSI 2003*, Paris, Sept. 2003.

[7] D. Gnaedig, M. Lapeyre, F. Mouchoux, E. Boutillon, “Efficient SIMD technique with parallel Max-Log-MAP algorithm for turbo decoders”, *Global Signal Processing Conference (GSPx)*, Sept. 2004.

[8] M. Arzel, C. Lahuec, F. Seguin, D. Gnaedig, M. Jézéquel, “Analog slice turbo decoder”, in *Proceedings. of the International Symposium on Circuits and Systems*, ISCAS'05, Kobe, Japan, May 2005.

[9] P. Coussy, D. Gnaedig, A. Nafkha, A. Baganne, E. Boutillon, E. Martin, "A methodology for IP integration into DSP SoC: a case study of a MAP algorithm for turbo decoder", in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'04, Montreal, Canada, May 2004.

Patents:

[10] E. Boutillon, P.Glenn Gulak, V. Gaudet, D. Gnaedig, "Procédé de codage et/ou de décodage de codes correcteurs d’erreurs, dispositifs et signal correspondants", Brevet français N° 0204764, France, Université de Bretagne Sud, Avril 2002.

[11] J. Tousch, E. Boutillon, D. Gnaedig, "Procédé et dispositif de décodage de codes par propagation de messages conjoints et systèmes les mettant en oeuvre", Brevet N°0406687000, TurboConcept, 17 juin 2004.

Contents

Chapter 3

Chapter 4

List of Figures

List of Tables

Notations

Throughout this work the notations of the matrices and the vectors are set in boldface, for example **M, y** are matrices of vectors depending on the context, while *M* and *y* are scalars.

Error correcting code notations

C : Code.
A_C : Alphabet of C.
k_c : Dimension of C, number of information bits.
n_c : Length of C, number of coded bits.
R : Rate of of C, $R = k_c / n_c$.
ν : Memory of the convolutional code.
K_c : Constraints length of the convolutional code $K_c = \nu + 1$.
m : Number of input bits of a convolutional code.
n : Number of output bits of a convolutional code, $n \geq m$.
N : Number of m-binary information symbols of a terminated convolutional code.
$\mathbf{S}^c$: Circulation state.
$\mathbf{G}$: Generator matrix of C.
σ : Noise variance
E_b : Energy per information bit.
N_0 : Noise real power spectrum density.
d, $\mathbf{d}$: Information message.
c, $\mathbf{c}$: Codeword.
u, $\mathbf{u}$: Modulated codeword.
v, $\mathbf{v}$: Received message.
$\hat{d}$: Decoded information message.
$\hat{c}$: Decoded codeword.
s_k, $\mathbf{S}_k$: Convolution encoder state at time index k.
δ : Scalar value of a m-binary symbol, $\delta = 0, \ldots, 2^m\text{-}1$
$\hat{\delta}$: Hard decision symbol.
$P_k(\delta)$: *A posteriori* probability for symbol δ at stage k.
$P_k^s(\delta)$: Probability of observation on the systematic bits, for a symbol δ at stage k.
$P_k^r(\delta)$: Probability of observation on the redundancy bits, for a symbol δ at stage k.
$P_k^a(\delta)$: *A priori* probability for symbol δ at stage k.
$P_k^e(\delta)$: Extrinsic probability for symbol δ at stage k.
$\alpha_k(s)$: Forward state probability of state s at stage k.
$\beta_k(s)$: Backward state probability of state s at stage k.
$\gamma_k(s, s')$: Transition probability between state s of stage k and state s' of stage k+1.
$\gamma_k^s(s, s')$: Transition probability considering only the systematic bits and *a priori* probability.
$\gamma_k^e(s, s')$: Transition probability considering only the redundancy bits.

$a_k(s)$	: Forward state metric of state *s* at stage *k*.
$b_k(s)$	: Backward state metric of state *s* at stage *k*.
$c_k(s,s')$	: Branch metric between state *s* of stage *k* and state *s'* of stage *k*+1.
$c_k^s(s,s')$	: Branch metric considering only the systematic bits and *a priori* probability.
$c_k^e(s,s')$	: Branch metric considering only the redundancy bits.
$L_k(\delta)$	: *A posteriori* Likelihood of symbol δ at stage *k*.
$L_k^a(\delta)$	: *A priori* information of symbol δ at stage *k*.
$L_k^e(\delta)$	: Extrinsic information of symbol δ at stage *k*.
S	: Spread of an interleaver.
d_{sum}	: Summary distance of a secondary cycle.
S_B	: Sphere Bound.
w	: Hamming weight of a codeword.
$n(w)$	: Number of codewords with Hamming weight *w*.
d_{min}	: Minimal distance of *C*.
Π	: Turbo code interleaver.
Π_T, Π_S	: Temporal and spatial permutations of an MSTC.
λ	: Multiplicative parameter of an ARP interleaver.
μ	: Offset parameter of an ARP interleaver.
C_p	: Cycle period of an ARP interleaver.
A	: Circular shift amplitude of the spatial permutation, $A = \{A_0, A_1, \ldots, A_{P-1}\}$.
$\mathbf{T}$	: Representation matrix of the spatial permutation of a MSTC.
Q	: Width of the first level in the temporal permutation.

Mathematical notations

$\mathbb{R}$	: is the field of real numbers.
exp x	: stands for the exponential of *x*, $x \in \mathbb{R}$.
log x	: stands for the natural logarithm of $x > 0$, $x \in \mathbb{R}$.
$log_b\ x$	: stands for the logarithm to the base *b* of $x > 0$, $x \in \mathbb{R}$.
$\lceil x \rceil, \lfloor x \rfloor$	: stand for the ceilling and floor functions of *x*, $x \in \mathbb{R}$.
$\mathbf{I}$	: Identity matrix.
$\boldsymbol{M}^t$	: stands for the transpose matrix of **M**.
$\lvert x \rvert_M$	: stands for the norm $\lvert x \rvert_M = \min(x \bmod M, -x \bmod M)$.
$d_M(t_1,t_2)$	: stands for the distance $d_M(t_1,t_2) = \lvert t_1 - t_2 \rvert_M$.
$\lVert k_1 - k_2 \rVert_M$	: stands for the distance between two symbols k_1 and k_2, that are not necessarily in the same slice.

Architecture notations

P	: Number of (SISO) processors.
K	: Number of slices for MSTCs.
M	: Trellis section size for parallel decoding, or slice size for MSTCs.
L	: Size of the window for a sliding window algorithm.
l	: Depth of the pipeline of a SISO processor.

$\mathbf{Z}_k^{(j)}$	: Extrinsic output of the decoder for trellis stage k at half-iteration j.
τ	: Time index.
Σ	: Schedule of the Max-Log-MAP algorithm.
I_{max}	: Maximal number of decoding iterations.
f_{clk}	: Clock frequency.
α	: Activity of the processing units.
D_S	: Input sample throughput of an architecture.
χ_{SM}	: Computational complexity of the state metric computation.
χ_{BM}	: Computational complexity of the branch metric computation.
χ_{SO}	: Computational complexity of the soft output computation.
χ_u	: Computational complexity of one stage of the Max-Log-MAP algorithm, $\chi_u = 2\cdot\chi_{SM}+2\cdot\chi_{BM}+\chi_{SO}$.
Γ	: Number of hardware computational resources of the architecture.
θ_Σ	: Number of additional cycles of schedule Σ. The total number of cycles for decoding M trellis stage is $M+\theta_\Sigma$.
β_Σ	: Maximal activity of schedule Σ.
φ	: Memory mapping function.
D_b	: Information bits throughput of a turbo decoder architecture.
d_P	: Degree of parallelism of a hybrid architecture, $d_P \leq P$.
m_a	: Number of accesses to the memory per cycle per processor.
m_p	: Number of ports of a memory bank.
ψ	: Average number of consistency conflicts for a transition of one half-iteration to the next one.
η	: Average proportion of consistency conflicts per transition.
$\Omega_{i\rightarrow n}$, $\Omega_{n\rightarrow i}$	: Overlapping regions for the transition from the interleaved to the natural order, and from the natural to the interleaved order, respectively.
$\notin_{i\rightarrow n}$, $\notin_{n\rightarrow i}$	: Banned regions corresponding to overlapping regions $\Omega_{i\rightarrow n}$ and $\Omega_{n\rightarrow i}$, respectively.
$\in$	: Interleaver mask for the constrained interleaver design.
Δ	: Overlapping depth for an overlapping decoding schedule.
Δ_{max}	: Maximal overlapping depth.
Λ	: Size of the secondary memory for the architecture with a hash-table.
h	: Hash-function.
w_d	: Number of bits used to represent a quantized version of a channel observation sample.
w_n	: Number of bits used to represent a quantized version of a states metric.
w_z	: Number of bits used to represent a quantized version of an extrinsic sample.

Acronyms

ACS	Add Compare Select.
ALU	Arithmetic and Logic Unit.
ARP	Almost Regular Permutation.
ASIC	Application Specific Integrated Circuit.
AWGN	Additive White Gaussian Noise.
BCJR	Bahl Cocke Jelinek Raviv (algorithm).
BER	Bit Error Rate.
CCSDS	Consultative Committee for Space Data Systems.
CRSC	Circular Recursive Systematic Convolutional (code).
DSP	Digital Signal Processor.
DVB	Digital Video Broadcasting.
DVB-RCS	Digital Video Broadcasting - Return Channel by Satellite.
EIM	Error Impulse Method.
FA	Full Adder.
FER	Frame Error Rate.
FPGA	Field Programmable Gate Array.
HI	Hierarchical Interleaver.
LDPC	Low Density Parity Check (code).
LE	Logic Element.
LIFO	Last In – First Out.
LSB	Least Significant Bit.
LUT	Look-Up Table.
MAG	Memory Address Generator.
MAP	Maximum A Posteriori.
MB	Memory Bank
ML	Maximum Likelihood.
MSB	Most Significant Bit.
MSTC	Multiple Slice Turbo Code.
NII	Next Iteration Initialization (technique).
NRZ	Non-Return to Zero (modulation).
PEP	Primary Error Pattern.
PRU	Pipelined Recursion Unit.
RAM	Random Access Memory.
ROM	Read Only Memory.
RSC	Recursive Systematic Convolutional (code).
RTZ	Return To Zero (sequence).
RU	Recursion Unit.
SEP	Secondary Error Pattern.
SIMD	Single Input Multiple Data.
SISO	Soft-Input Soft-Output (decoder).
SMU	State Metric Update.
SNR	Signal to Noise Ratio.
SOVA	Soft Output Viterbi Algorithm.
SP	Spatial Permuter.
TPC	Turbo Product Code.
UMTS	Universal Mobile Telecommunication System.
VLSI	Very Large Scale Integration.

Introduction and contributions

Introduction

The development of telecommunications and especially the explosion of "personal" communications (wireless indoor and outdoor communications, long and short distance transfers of personal data, home networks, etc.) pose a technical challenge for the reliable transmission of information. Rigorous studies in the field of information transmission started with the work of Shannon in 1948. Shannon demonstrated that reliable information transmission was possible when the information data rate is lower than the capacity of the transmission channel. This objective is achieved thanks to the use of error-correcting codes, which makes it possible to correct the transmission errors. This promised limit was the target to reach for the many error-correcting code constructions that were designed during the 40 years that followed, yet none of them approached the Shannon limit by less than 2 dB. An important breakthrough for approaching the channel capacity was achieved only one decade ago with the invention of turbo codes by Claude Berrou and Alain Glavieux in 1993. Turbo codes made it possible to get within a few tenths of a dB from the Shannon capacity, for a bit error rate of 10^{-5}.

The principle of turbo coding relies on a simple widely held concept involving the decomposition of a complex problem into several sub-problems that are easier to solve. Turbo codes are actually designed as the association of two simple codes linked together by a temporal permutation function (called interleaver), and the decoders associated with the two codes collaborate together by exchanging information iteratively in order to correct the transmission errors. The iterative decoding principle makes it possible in practice to implement a real-time decoder adapted to such a complex and powerful code using the present capabilities of the technology. The excellent performance of turbo codes and their suitability for practical implementations explain their introduction into communications standards for spatial (*e.g.* CCSDS, DVB-RCS), cellular wireless (*e.g.* UMTS) and recently broadband applications (*e.g.* WiMAX). The first applications employing the turbo code technology required only a few units of Mbits per second to be transmitted (often much less), thus decoder implementation is well mastered today. But, along with the recent introduction of turbo codes into the IEEE 802.16 standard (WiMAX), there is an increasing demand for high-speed turbo decoder implementation up to a few hundred Mbits/s. The design of hardware architecture for high-speed turbo decoding has thus become a concrete issue for industry and a real challenge for researchers.

Our work concentrates on convolutional turbo codes, and on their high-throughput iterative decoding. Since the problems of finding a powerful turbo code and efficient decoding implementation are closely linked together, we focus both on the design of codes and especially interleavers, and on high-speed hardware decoding architectures, which justifies the title of the manuscript. In our work on this subject, we study a large spectrum of possibilities that are available in the design space, ranging from the decoder design for existing and fixed coding structures to the joint design of a powerful code and the associated

high-speed decoder architecture. The objective is to design efficient turbo decoder architectures presenting the most advantageous trade-off in terms of speed and error rate performance versus complexity. The organization of the manuscript is as follows.

The first chapter is a presentation of the context of our work: error correcting codes and especially convolutional codes and turbo codes. Since double-binary turbo codes using tail-biting constituent codes are mainly used in this thesis, our description concentrates on Circular Recursive Systematic Convolutional (CRSC) codes. The corresponding iterative decoding algorithm using Soft-Input Soft-Output (SISO) decoders is also derived. The derivation of the Maximum A Posteriori (MAP) SISO decoding algorithm will lead to its simplest expression: the Max-Log-MAP algorithm. Sliding window execution schedules of this algorithm are also presented.

The second chapter introduces the basic concepts of the architecture design. In particular, our contribution includes the introduction of an analytical model for the expression of the throughput of a given SISO architecture and its efficiency. The expression of the throughput will exhibit a product of three factors, each one corresponding to a possible direction for improvement: the number of processing units, the average ratio of utilization of these processing units and the clock frequency. This average ratio is called the activity of the processing units and is a crucial consideration especially for parallel implementations. The application of this model to turbo decoder architectures is then discussed and exhibits an important potential for improvement compared to state-of-the-art architectures. After having identified the three major research fields - namely parallel decoding, augmentation of the activity and clock frequency increase – we try to improve each of them in the three following chapters in order to improve the overall turbo decoder throughput.

The third chapter concerns the parallel decoding of turbo codes using several SISO processors working in parallel. It addresses the main bottleneck in parallel architectures: concurrent accesses to the memory. This issue is related to the presence of the interleaver in the coding scheme. Therefore, our contribution relies on a joint code-architecture design that solves this issue using Multiple Slice Turbo Codes (MSTCs). After a description of this new family and of their design, alternative constrained interleaver designs proposed in the literature are also discussed.

The fourth chapter focuses on the improvement of the efficiency of the architecture characterized by the activity of the processing units. This concern is even more critical in the context of parallel decoding. Several solutions are explored: the proposition of a new schedule for parallel decoding, hybrid parallel/serial architectures and overlapping of consecutive half-iterations. This latter solution leads to another issue that is resolved thanks to a joint interleaver-architecture design again using MSTCs. The results presented in this chapter have not yet been published.

The fifth chapter presents a technique for increasing the clock frequency of the architecture. In SISO decoder the clock frequency is limited by the critical path in the feedback loop of the recursive forward and backward units: the "Add-Compare-Select

bottleneck". Our solution is adapted to parallel decoding and in particular to double-binary convolutional codes. This solution enables us to almost double the throughput of the architecture, at no complexity cost, on a Field Programmable Gate Array (FPGA) implementation. The results presented in this chapter have yet not been published.

The last chapter applies some key contributions of this thesis to a turbo decoder implementation for the WiMAX standard. The number of processors is scalable from 1 to 4. The design is application-oriented, covering a large range of frame sizes and code rates. Synthesis results on an FPGA as well as error correcting performance are given.

Finally, in **the conclusion,** we summarize the results obtained and give some **perspectives** for further research in the domain of efficient architectures for iterative decoding schemes.

In addition, a thorough state-of-the-art concerning the high-speed decoding of convolutional turbo codes is presented throughout the present manuscript.

Contributions of this thesis:

- A simplified architectural model for the throughput and efficiency of turbo decoder architectures.
- Design and optimization of a new class of turbo codes: Multiple Slice Turbo Codes (MSTCs).
- Design and optimization of three-dimensional MSTCs (presented in Appendix D).
- Establishment of an upper bound of the spread of an interleaver for multi-dimensional turbo codes and MSTCs (summarized in Appendix B).
- A joint interleaver-architecture design method suitable for parallel decoding using MSTCs.
- A new schedule for the MAP algorithm adapted to parallel decoding with at least two processors, increasing the SISO processor activity.
- A hybrid architecture combining the advantages of parallel and serial architectures.
- An approach for maximizing the activity of the architecture in the context of iterative decoding, involving the overlapping of consecutive half-iterations. This solution leads to the utilization of extrinsic information that are not up-to-date and that we have called a consistency conflict.
- An analytical evaluation of the average number of consistency conflicts for a uniform interleaver.
- A joint interleaver-architecture design methodology based on MSTCs preventing consistency conflicts.

- A pipeline multiplexing technique adapted to parallel decoding that is well suited to double-binary convolutional codes.
- Application of MSTCs for the implementation on a Digital Signal Processor with Single Instruction Multiple Data capabilities (presented in Appendix E).

Other contributions not presented in this manuscript:

- A performance/complexity comparison of SISO decoding algorithms for double-binary RSC codes: the Max-Log-MAP algorithm and the Log-MAP algorithm working in the dual domain.
- A methodology based on the Simplex algorithm for optimizing the iteration-dependent factors used to scale the extrinsic information obtained with the Max-Log-MAP algorithm.
- A methodology based on the Error Impulse Method for selecting the parameters of an Almost Regular Permutation (ARP) interleaver using a gradual "test, try and select" approach.
- A study of the utilization of stopping criterions associated with MSTCs in order to reduce the complexity of the decoder.

Chapter 1

Context and state of the art

Contents:

Chapter 1. Context and state of the art

This first chapter introduces the basic concepts of error correcting codes, turbo coding and turbo decoding.

The first section gives an overview of the fundamental concepts of channel coding and introduces error-correcting codes. The second section focuses more precisely on convolutional codes and especially Circular Recursive Systematic Convolutional (CRSC) codes that will be used throughout this manuscript.

Then, turbo codes are introduced in the third section as a parallel concatenation of Recursive Systematic Convolutional (RSC) codes, separated by an interleaver. The importance of this component and its design will also be discussed.

The iterative decoding principle of turbo codes involves soft-input soft-output (SISO) decoders, whose algorithms are described and compared for double-binary turbo codes in the fifth section. The expression of the optimal SISO algorithm, the BCJR algorithm is presented and simplified into its logarithmic expression, the Max-Log-MAP algorithm, which is suitable for practical implementations. After an evaluation of the computational complexity of this algorithm, its decoding scheduling is discussed. This will lead us to the introduction of sliding window schedules that will be simplified to their minimal expression in order to suppress all unnecessary computations.

1.1 Fundamentals of channel coding

1.1.1 Coding theory

Coding theory started with the pioneering work of C. Shannon (1948), who established the fundamental problem of reliable transmission of information. Shannon introduced the channel capacity $\mathcal{C}$ (expressed in bits per channel use) as a measure of the maximum information that can be transmitted over a channel with additive noise. His main theorem is as follows: if the information rate $\mathcal{R}$ from the source is less than the channel capacity $\mathcal{C}$, then it is theoretically possible to achieve reliable (error-free) transmission over the channel by an appropriate coding [Shannon *et al.*-48]. Using the set of random codes, this theorem proved the existence of a code enabling information to be transmitted with any given probability of error. But, it does not give any idea about how to find such a code achieving the channel capacity.

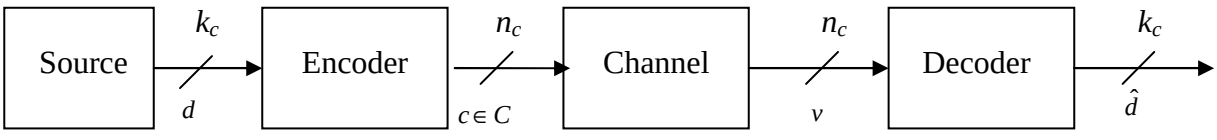


Figure 1-1: Transmission scheme using an error correcting code

The work of Shannon stimulated the coding theory community to find error correcting codes capable of achieving the channel capacity. With such a code C, a message d of k_c information symbols is encoded through an encoder providing a codeword c of C, composed of n_c coded symbols c_k, $k = 0,\ldots,n_c\text{-}1$ (see Figure 1-1) taken from an alphabet A_C. The ratio $R = k_c / n_c$ denotes the code rate. After transmission over a noisy channel, the noisy symbols v are received by the decoder. This latter uses the properties of the code to try to recover the transmission errors by finding the most probable transmitted codeword $\hat{c}$, or equivalently the most probable message $\hat{d}$.

In order to assess the performance of a code in terms of Bit Error Rate (BER) or Frame Error Rate (FER), analytical bounding can be performed [Huffman *et al*.-03], derived from the knowledge of the code properties. Alternately, the BER (or FER) can be estimated through a Monte-Carlo simulation method using a given decoder. This latter solution has the advantage (or disadvantage) of characterizing the code and the decoding algorithm together.

1.1.2 Optimal decoding

The optimal error decoding performance is obtained for a decoder satisfying the *Maximum A Posteriori* (MAP) criterion. It aims to find the most probable codeword $\hat{c}$ based on the channel output v, and on the knowledge of the code C:

$$\hat{c} = \arg\max_{c' \in C} \Pr(c = c' \mid v). \quad \textbf{(1-1)}$$

Using Bayes' rules, the *posterior probability* $\Pr(c = c' \mid v)$ is calculated as:

$$\Pr(c = c' \mid v) = \frac{p(v \mid c = c')\Pr(c)}{p(v)}. \quad \textbf{(1-2)}$$

If the symbols of the sources are equally probable, the MAP criterion involves the maximization of the conditional probability $p(v \mid c = c')$ expressed as

$$\hat{c} = \arg\max_{c' \in C} p(v \mid c = c'), \quad \textbf{(1-3)}$$

which corresponds to a Maximum Likelihood (ML) decoding. The conditional probability $p(v \mid c = c')$ represents the probability of the observation v at the output of the channel given that the transmitted codeword is c'. Its computation is described in the next section.

This expression of the MAP criterion leads to a word decoding algorithm minimizing the FER. To minimize the BER, the MAP criterion is applied symbol-by-symbol for each codeword symbol c_k, using the following decision rule:

$$\hat{c}_k = \arg\max_{c' \in A_C} \Pr(c_k = c' \mid v). \quad \textbf{(1-4)}$$

1.1.3 Channel model

The channel depicted in Figure 1-1 corresponds to the succession of three components: the modulator, the transmission link and the demodulator. The modulator transforms the digital representation of the codeword into an analog signal that is transmitted over the link to the

receiver. At the receiver side, the demodulator converts the analog signal into its digital counterpart that is fed into the decoder. Assuming perfect modulation, synchronization and demodulation, the channel is represented by a discrete-time equivalent model.

In our study, this model is represented by a binary-input additive white Gaussian noise (AWGN) of mean 0 and variance σ^2 using a binary phased shift keying (BPSK) modulation, where $c_k \in \{0,1\}$ are mapped onto modulated symbols $u_k \in \{-1,+1\}$. The modulated codeword is denoted u. We also assume that the channel is memoryless. Thus, the conditional probability is expressed as

$$p(v|c) = p(v|u) = \prod_{k=0}^{\eta-1} p(v_k|u_k), \quad \textbf{(1-5)}$$

where $p(v_k|u_k)$ is calculated using

$$p(v_k|u_k) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(v_k-u_k)^2}{2\sigma^2}\right). \quad \textbf{(1-6)}$$

The BER and FER performance is generally given as a function of the signal to noise ratio E_b / N_0, where E_b is the energy per information bit and N_0 is the real power spectrum density of the noise. The variance of the channel is thus expressed as a function of the signal to noise ratio and of the code rate R using:

$$\sigma^2 = \frac{1}{2R \cdot \frac{E_b}{N_0}}. \quad \textbf{(1-7)}$$

We refer the reader interested in further details about channel models to [Proakis].

1.1.4 Linear block codes

The first error correcting codes discovered were linear block codes. The underlying mathematical theory of these codes uses the notion of Galois Fields (GF).

A (n_c,k_c) linear block code C with symbols in the Galois Field GF(2^q) is a k_c-dimensional subspace of the n_c-dimensional vector space of $GF(2^q)^{n_c}$. The code C can be defined as the list of all codewords: $C = \left\{(c_0, c_1, ...c_{\eta-1}), c_j \in GF(2^q)\right\}$. The parameter n_c is called the length of the code, and k_c is called the dimension. The rate of the code is defined as $R = k_c / n_c$. Given the information vector d, the codeword c is obtained with the linear expression $d = c \cdot \mathbf{G}$, where $\mathbf{G}$ is any $k_c \times n_c$ matrix whose rows form a basis for C. Such a matrix is called the generator matrix of C.

Each code C is associated with its dual code $C^\perp$, defined as the set of codewords $c^\perp$ such that there exists a codeword c of C orthogonal with $c^\perp$:

$$C^\perp = \left\{c^\perp \middle| \exists c \in C, c.c^\perp = 0\right\} \quad \textbf{(1-8)}$$

The dual code has length n_c, dimension n_c - k_c and rate $R = (n_c - k_c)/ n_c$. The dual code of C is generated by a $(n_c - k_c) \times n_c$ generator matrix $\mathbf{H}$ verifying $\mathbf{G} \cdot \mathbf{H}^t = 0$. An important property

of the dual code that will be used later on is: for a code C of rate above 1/2, the dimension of the dual code $C^{\perp}$ is lower than the dimension of C.

For a binary code, the Hamming weight $w(c)$ of a codeword c is defined as the number of ones in the codeword, and the distance between two codewords c and c' is defined as the difference of their Hamming weight $d_w = w(c) - w(c')$. The minimum distance d_{min} of C is defined as the minimum distance between two codewords of C. For a linear code, the minimum distance corresponds to the minimum Hamming weight among all the codewords of C:

$$d_{\min} = \min_{c \in C} w(c) \qquad \textbf{(1-9)}$$

Parallel to the development of powerful linear block codes, another class of error-correcting codes has been introduced based on linear finite-state shift registers. They are called convolutional codes and described in the next section.

1.2 Convolutional codes

Binary convolutional codes were proposed by Elias in 1955 [Elias-55]. Today, in practical telecommunications systems, convolutional codes are widely used because of their high error correction capability and the low complexity of their encoding and decoding scheme.

In this section, we describe the basic principles of convolutional codes and their representations. This description will lead us to the introduction of Circular Recursive Convolutional codes, which are of particular interest in this manuscript.

1.2.1 Definition

A rate m/n convolutional code C is a linear application that transforms blocks of m symbols, provided by the source of information, into blocks of n symbols ($n > m$). These information symbols taken from an alphabet A_C are also called messages. The transformation is a function of the last K_C input blocks, where K_C is the constraint length of the convolutional code C. The value $\nu = K_C - 1$ denotes the memory of the convolutional code, which is therefore noted (ν, m, n). The main difference between a convolutional code and a block code lies in this memory effect introduced by convolutional encoding.

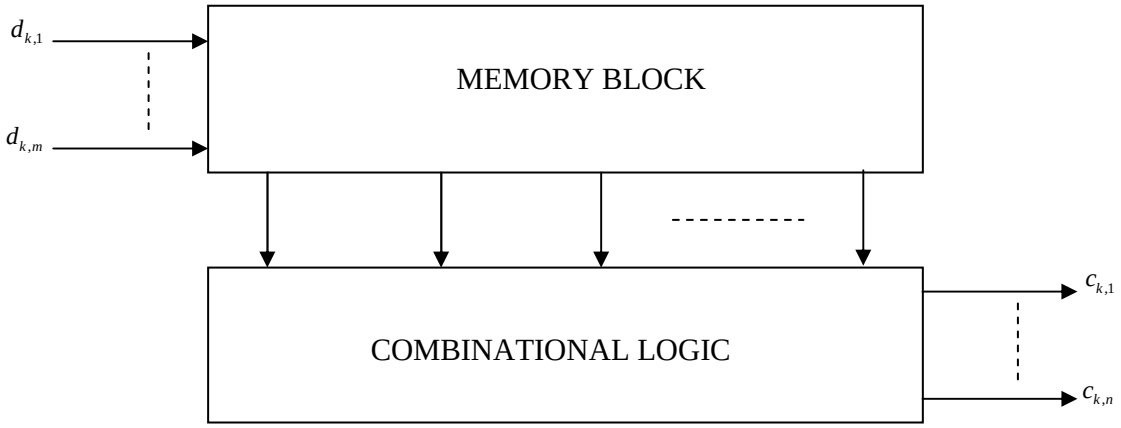


Figure 1-2: General principle of convolutional encoding

More precisely, a convolutional encoder receives a possibly semi-infinite sequence d of information blocks $d = (\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_k, \ldots)$ starting at a finite time $k = 0$, and the sequence may or may not end. Each block $\mathbf{d}_k$ has m symbols $\mathbf{d}_k = (d_{k,1} \cdots d_{k,l} \cdots d_{k,m})^T$ taken from A_C. The information sequence is fed into the convolutional encoder in blocks, one after the other: at time k, block $\mathbf{d}_k$ is fed into the encoder and a block $\mathbf{c}_k$ of n symbols is produced, $\mathbf{c}_k = (c_{k,1} \cdots c_{k,l} \cdots c_{k,n})^T$. Thus the encoder output sequence is $c = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k, \ldots)$, which is considered as one codeword of C. Another way of representing the input and output sequences uses the delay operator D, introduced by Forney [Forney-70]. With these notations, the sequences d and c are written as: $d(D) = \mathbf{d}_0 + \mathbf{d}_1 D + \mathbf{d}_2 D^2 + \ldots$ and $c(D) = \mathbf{c}_0 + \mathbf{c}_1 D + \mathbf{c}_2 D^2 + \ldots$. The exponent of the operator D shows at which time the coefficient $\mathbf{d}_k$ or $\mathbf{c}_k$ appears in the sequence.

Each output block $\mathbf{c}_k$ is computed as a linear combination of the last K information blocks $\mathbf{d}_k$, $\mathbf{d}_{k-1}$, $\mathbf{d}_{k-2}$,…, $\mathbf{d}_{k-\nu}$:

$$\mathbf{c}_k = \mathbf{d}_k \left(\mathbf{G}_0 + D \cdot \mathbf{G}_1 + D^2 \cdot \mathbf{G}_2 + \ldots + D^\nu \cdot \mathbf{G}_\nu \right), \qquad \textbf{(1-10)}$$

where each $\mathbf{G}_i$ is a $m \times n$ matrix over A_C called a generator sub-matrix. The $m \times n$ matrix defined as $\mathbf{G}(D) = \mathbf{G}_0 + \mathbf{G}_1 D + \ldots + \mathbf{G}_\nu D^\nu$ is called the polynomial generator matrix. (1-10) is then expressed as

$$c(D) = d(D)\mathbf{G}(D). \qquad \textbf{(1-11)}$$

Convolutional codes can be classified following two main characteristics: first the systematic or non-systematic attribute, and secondly the recursive or non-recursive attribute. For a systematic code, the coded sequence contains the information sequence. Up to now, only non-recursive codes have been consider. Recursive codes will be described in the next section.

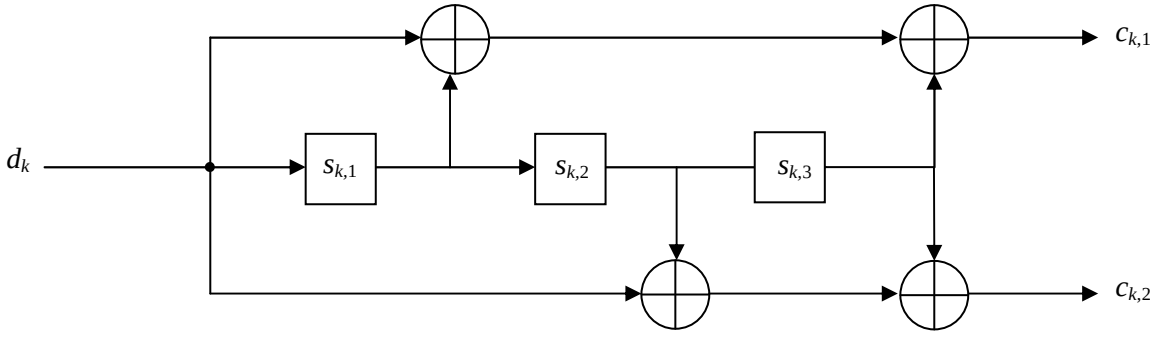


Figure 1-3: Convolution encoder (3,1,2) with generator polynomials (15,13)

Example 1-1:

Let us consider a binary systematic convolutional code of rate $R = 1/2$ and memory $\nu = 3$ as depicted in Figure 1-3. The memory is implemented through a shift register, receiving the input messages in blocks of one bit. The combinational logic, which outputs blocks of two bits, is implemented with *exclusive or* gates (XOR), which is equivalent to the sum operator in the field GF(2). It transforms the input sequence into the output sequence using the following relations:

$$\mathbf{c}_{k,1} = d_{k,1} + d_{k-1,1} + d_{k-3,1} \text{ and } \mathbf{c}_{k,2} = d_{k,1} + d_{k-2,1} + d_{k-3,1}, \quad \textbf{(1-12)}$$

where the additions are done in GF(2).

Let $d_1(D)$, $c_1(D)$ and $c_2(D)$ denote the formal series corresponding to the sequences $\left(d_{k,1}\right)_{k\geq 0}, \left(c_{k,1}\right)_{k\geq 0}$ and $\left(c_{k,2}\right)_{k\geq 0}$. The input sequence is then written as:

$$d_1(D) = \sum_k d_{k,1} D^k \quad \textbf{(1-13)}$$

Then the relation (1-12) can be rewritten as:

$$c_1(D) = d_1(D)\left(1 + D + D^3\right) \text{ and } c_2(D) = d_1(D)\left(1 + D^2 + D^3\right) \quad \textbf{(1-14)}$$

By identification with (1-11), the polynomial generator matrix is then:

$$\mathbf{G}(D) = \left(1 + D + D^3 \quad 1 + D^2 + D^3\right) \quad \textbf{(1-15)}$$

A binary representation ($G_{c_1} = 1101_2$ and $G_{c_2} = 1011_2$) of the generator polynomials $G_{c_1}(D) = 1 + D + D^3$ and $G_{c_2}(D) = 1 + D^2 + D^3$, respectively, specifies the values of the coefficients ordered following the increasing powers of *D*. The representation of these generators in octal format ($G_{c_1} = 15_8$ and $G_{c_2} = 13_8$) is commonly used, and the notation is then simplified to (15,13).

1.2.2 Representation diagrams

The evolution of a binary convolutional encoder can be represented as a finite state machine, whose outputs are a function of the current state and of its inputs. The state of this machine corresponds to the ν bits of the shift register memory, which yields 2^ν distinct states.

Let us consider $\mathbf{S}_k = \left(s_{k,1} \quad s_{k,2} \quad \ldots \quad s_{k,\nu}\right)^T$ the vector representing the encoder state at time *k*, where $s_{k,i}$ represents the value of the *i*-th register of the encoder at time *k* (see Figure 1-3). For convenience, the encoder state $\mathbf{S}_k$ is also denoted by the scalar quantity

$$s_k = \sum_{i=1}^{2^\nu} 2^{i-1} s_{k,i}, \quad \textbf{(1-16)}$$

taking values between 0 and 2^ν-1, and is then written as $\mathbf{S}_k \equiv s_k$.

State diagram:

The first diagram representing the evolution of the finite state machine is the state diagram. It represents all 2^ν states of the encoder, and also the transitions between them. The input bit of the encoder determines the transition between states. The outputs depend on the current state and on the input bit. Each transition in the state diagram is represented by the couple input/outputs ($d_k/c_{k,1}c_{k,2}$). The state diagram of Example 1-1 is depicted in Figure 1-4.

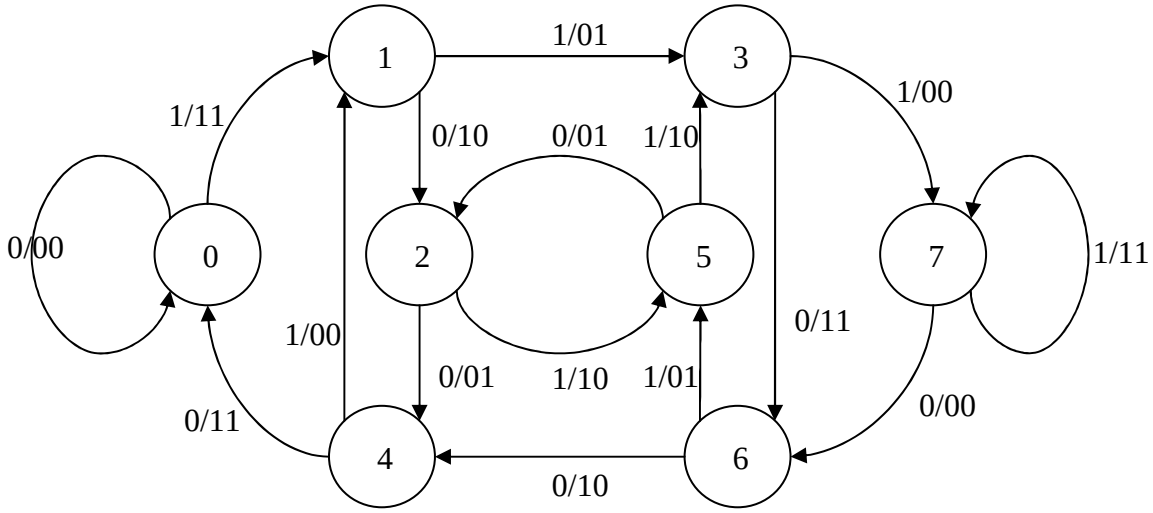


Figure 1-4: State diagram associated with the encoder (3,1,2) with generator polynomials (15,13)

Trellis diagram:

The trellis diagram introduced in [Forney-73] is the most commonly used diagram for illustrating the decoding principle of convolutional codes. The trellis diagram for the convolutional code (3, 1, 2) presented in Figure 1-3 is represented in Figure 1-5.

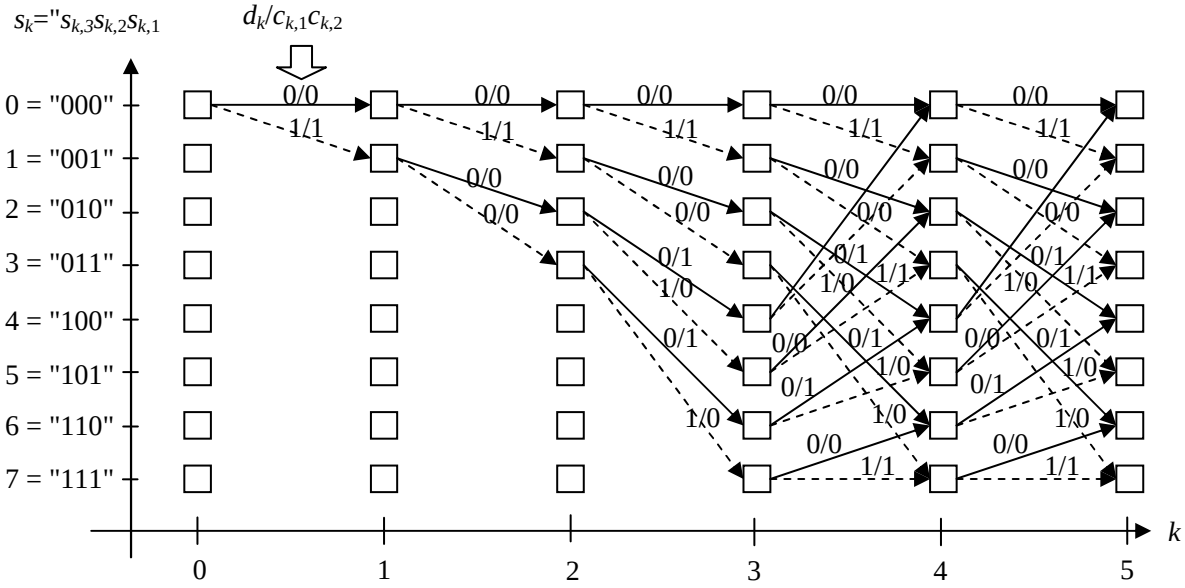


Figure 1-5: Trellis diagram associated with the encoder (3,1,2) with generator polynomials (15,13)

The *x*-axis represents the discrete time *k* and the *y*-axis the 2^{ν} states of the convolutional encoder. The induced discrete points correspond to the possible successive states of the encoder. The transitions between the states are represented by branches oriented from left to right. The trellis diagram is thus constructed by connecting through the branches all possible evolutions from one state to another.

The time evolution of the encoder, for a given information sequence, is viewed on the trellis diagrams by a succession of states connected through the branches. This connected

sequence is called a path in the trellis. A sequence of information and coded bits is associated with each path. The decoder uses observations on the coded sequence and tries to recover the associated path in the trellis, thus yielding the decoded information sequence.

1.2.3 Recursive Systematic Convolutional codes

An important class of convolutional codes of special interest for turbo codes is the class of Recursive Systematic Convolutional codes (RSC). RSC codes introduce feedback into the memory of convolutional codes.

A RSC code is obtained by introducing a feedback loop in the memory (recursive) and by transmitting directly the input of the encoder as an output (systematic). For an RSC code, the input of the shift register is a combinatory function of the input block $\mathbf{d}_k$ and its delayed versions $\mathbf{d}_{k-1}$, $\mathbf{d}_{k-2}$,…, $\mathbf{d}_{k-v}$. It is worth noting that the set of codewords generated by the RSC encoder is the same as that generated by the corresponding non systematic encoder. However, the correspondence between the information sequences and the codewords is different.

Example 1-2:

In practice, a binary RSC code is obtained from a binary non-systematic non-recursive convolutional code by replacing the input of the linear shift register by one of the outputs of the non systematic convolutional code. The corresponding output is replaced by the input of the encoder. This construction leads to an implementation of a Linear Feedback Shift Register (LFSR) as represented in Figure 1-6.

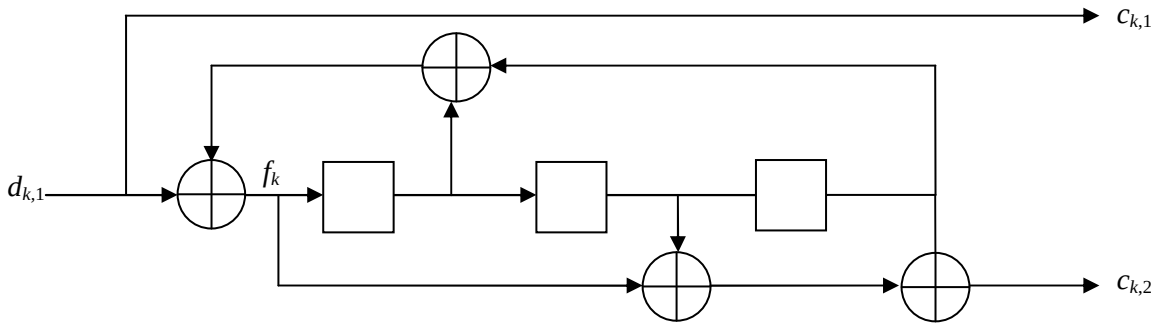


Figure 1-6: Recursive systematic encoder (3,1,2) generator polynomials (15,13)

Using the convolutional code of Example 1-1, the non-systematic convolutional encoder is transformed into the corresponding RSC encoder depicted in Figure 1-6. The polynomial generator matrix of the RSC encoder is obtained from that of the non-systematic encoder (1-15) by dividing $\mathbf{G}$ by $G_{c_1}(D)$:

$$\mathbf{G}_{RSC}(D) = \left(1 \quad \frac{1+D^2+D^3}{1+D+D^3}\right) \tag{1-17}$$

RSC codes are used as component codes for turbo codes, because of their infinite impulse response: if a binary RSC encoder is fed with a single “1” and then only with “0”, due to the feedback loop, the weight of the parity sequence produced is infinite. For a non recursive

encoder, only a maximum of K_C "1" bits are generated, then the "1" comes out of the shift register.

Example 1-3:

Let us consider the binary RSC code (3,2,3) depicted in Figure 1-7 and used throughout this manuscript. This code is called a double-binary[1] convolutional code, since the encoder is fed with two bits, which are encoded simultaneously.

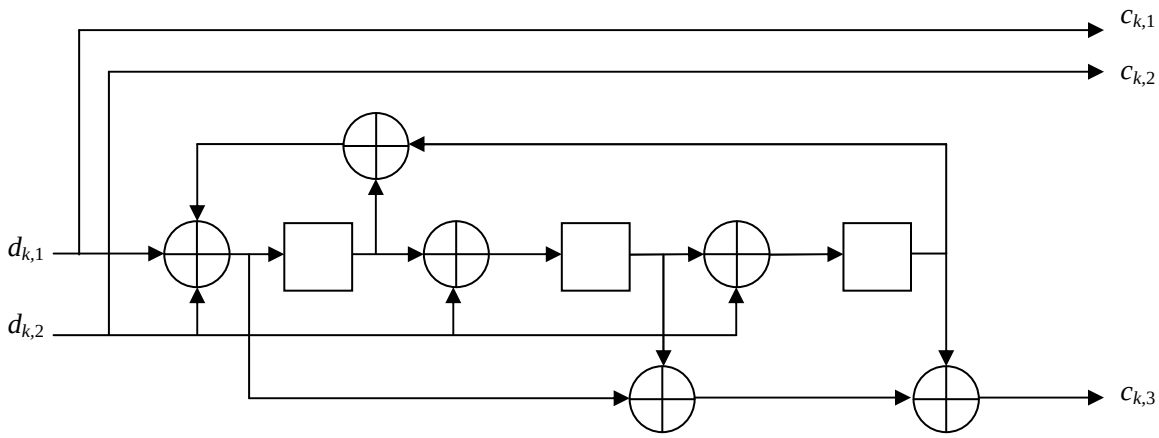


Figure 1-7: Description of the double-binary RSC encoder (3,2,4) with generator polynomials (15,13)

1.2.4 Punctured convolutional code

The rate of a convolutional code C with m input bits and n output bits is $R = m/n$. Higher rates can be achieved simply by discarding, or puncturing, some of the redundancy symbols. Thus, a single convolutional code C generates a whole set of convolutional codes with a large range of code rates R_p. Another advantage of this solution is that the same decoder can be used to decode the set of punctured codes derived from the single mother code C [Hagenauer–88]. In practice, puncturing can be performed at the output of an encoder by discarding the encoded symbols according to a puncturing pattern that is repeated periodically. In the general case, both the information symbols and redundancy symbols can be punctured, but we restrict ourselves in the sequel to puncturing the redundancy symbols.

For example, the mother code of rate 1/2 depicted in Figure 1-6 is punctured to generate several different code rates. Table 1-1 gives some rates and the corresponding puncturing patterns that can be achieved by puncturing the redundancy bit $\mathbf{c}_{k,2}$. A "0" means that the symbol is discarded, while a "1" means that the symbol is transmitted. The puncturing pattern is repeated periodically.

R_p	1/2	2/3	3/4	4/5	5/6
Puncturing pattern	1	10	100	1000	10000

Table 1-1: Puncturing patterns and codes rates R_p obtained by puncturing the 1/2 RSC code of Figure 1-6

[1] The denomination duo-binary denotes a modulation scheme derived from Non-Return to Zero (NRZ) modulation. Since the usage of this denomination for RSC codes in previous papers led to confusion, it has been replaced by the term double-binary. Note, however, that the code is binary.

1.2.5 Terminated convolutional codes

In this manuscript, the case of terminated (or finite) convolutional code of size N blocks is considered. A finite convolutional code of size N and rate $R = m / n$ is thus a linear block code of dimension $N \cdot m$ and length $N \cdot n$, denoted $C(N \cdot n, N \cdot m)$.

For a finite convolutional code, the initial state of the shift register is generally set to the "all zero" state by the encoder. This information is used by the decoder to enhance the decoding performance. Nevertheless, since the final state of the encoder is undetermined and depends on the input sequence, the decoder has no special information available regarding the final state of the shift register. Several alternatives to terminate the convolutional code have been proposed in the literature:

- No termination: the final state of the encoder, which depends on the input sequence, is not known at the decoder side. This lack of information leads to a degradation in performance for the last encoded data and therefore to less effective error protection. In particular, the last encoded symbols are associated with codewords with low Hamming weights, resulting in a reduction of the asymptotic gain of the code. This degradation is increased when the block size N increases.
- Forcing the encoder state at the end of the encoding phase to a known final state. Generally, the final state is forced to the "all zero" state by encoding $N \cdot m$ symbols and feeding the shift register with the appropriate sequence of symbols, called tail symbols. The resulting encoded block is of length $(N+\nu) \cdot m$. On the one hand, the initial and final states of the shift register are known by the decoder. On the other hand, the spectral efficiency of the transmission is decreased and the *rate loss* $\nu/(\nu+N)$ due to these tail symbols is increased as the block length N is shorter.
- Using a circular code. With circular convolutional codes, the encoder is not initialized to the "all zero" state but at a given initial state, and the encoder retrieves this initial state at the end of the encoding process. This initial state called "circulation state" depends on the generator polynomials of the code and also on the input sequence of the encoder. The technique of circular encoding was proposed in [Ma *et al*.-86] for non-systematic convolutional code (so called "tail-biting") and was adapted for recursive codes in [Berrou *et al*.-99a]. At the decoder side, the circulation state is not known, but the trellis can be seen as a circle, without any discontinuity in transitions between states. This property can be used by the decoder leading to an equal error protection of all symbols of the block.

To compute the output sequence of the circular convolutional code, two approaches have been proposed in the literature:

- The first solution proposed in [Ma *et al*.-86] and [Berrou *et al*.-99a] initializes the encoder at the circulation step $\mathbf{S}^c$ and performs a second encoding operation of the N data to produce the redundant bits.

- For the second solution developed by Pietrobon in [Pietrobon-00], the first encoding step also computes the circulation state. Moreover, the encoded sequence corresponding to the "all zero" initial state produced during this encoding step is stored. During the second encoding step, corresponding to a post-processing step, this encoded sequence is added to a fixed (non-data dependent) pattern depending on the circulation state $\mathbf{S}^c$ to produce the encoded sequence corresponding to the circulation state.

1.3 Turbo coding

After giving a brief overview of the concept of turbo coding, we describe in detail the turbo encoding principle. We then develop an example of a turbo code using a double-binary CRSC code that will be used throughout this manuscript. Finally, some basics about interleaver design are recalled.

1.3.1 Overview

A fundamental class of error correcting codes was proposed by C. Berrou and A. Glavieux in 1993 [Berrou *et al.*-93a]. These codes are constructed as a parallel concatenation of two recursive systematic convolutional codes separated by an interleaver. They are called "turbo codes" in reference to their practical iterative decoding principle, by analogy with the turbo principle of a turbo-compressed engine. This latter actually uses a retro-action principle by reusing the exhaust gas to improve its efficiency. The decoding principle is based on an iterative algorithm where two component decoders exchange information improving the error correction efficiency of the decoder during the iterations. At the end of the iterative process, after several iterations, both decoders converge to the decoded codeword, which corresponds to the transmitted codewords when all transmission errors have been corrected.

1.3.2 Turbo encoding principle

Let RSC_1 and RSC_2 be two identical *m/n* RSC encoders. A turbo code is constructed as a parallel concatenation of two RSC encoders, *i.e.* the input sequence $(\mathbf{d}_k)_{k\geq 0}$ is encoded twice: once by each RSC encoder. An interleaver Π (the interleaver is also referred to as a permutation) is performed on the information sequence prior to the RSC_2 encoding. This interleaver plays a crucial role in the construction of the code, because it performs a pseudo random permutation of the latter input sequence, which introduces randomness in the encoding scheme. In other words, the two constituent encoders code the same information sequence $(\mathbf{d}_k)_{k\geq 0}$ but in a different order. Randomness for error correcting codes is a key attribute enabling us to approach channel capacity. As discussed in the first section, the work of C. Shannon proved that it was possible to achieve channel capacity (the so-called "Shannon bound") using random codes. The Shannon theorem is a proof of the existence of such a code achieving channel capacity, though. Neither a code nor a decoding algorithm making it possible to achieve it had been proposed before. The efficiency of turbo codes and their iterative decoding principle described in [Berrou *et al.*-93a] made it possible for the first time to approach at 0.7 dB of the channel capacity, for a Bit Error Rate (BER) of 10^{-5}.

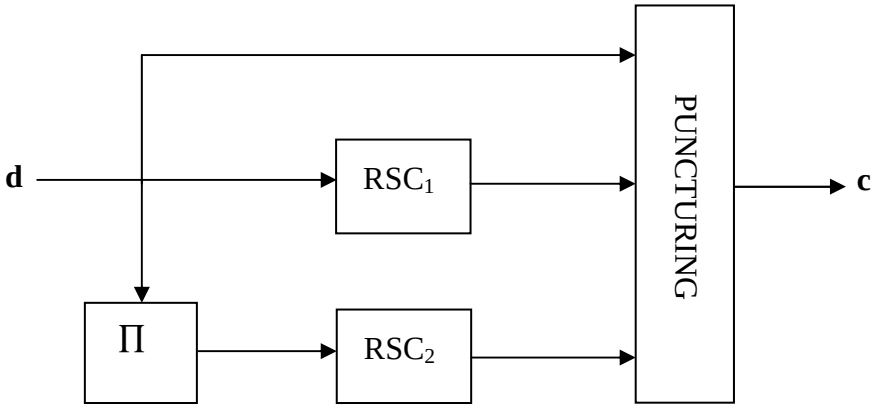


Figure 1-8: Description of the parallel concatenation of RSC codes separated by an interleaver

The first turbo codes were designed with non-terminated convolutional codes as component codes. This class of turbo codes are convenient for applications using continuous encoding and decoding schemes, like for instance broadcasting (*e.g.* digital video broadcasting). For interactive or multi-user applications transmission of blocks is required. Consequently, turbo codes using terminated RSC codes were adopted by the CCSDS for deep-space communications [CCSDS-99]. Turbo codes were then introduced in satellite communication systems such as Inmarsat, DVB-RCS [DVB-RCS]. The same choice was also made for the 3rd generation of mobile communications systems (UMTS [3GPP], CDMA2000 [3GPP2], etc.), and more recently of the WiMAX standard IEEE 802.16 [WiMAX].

Other turbo codes using serial concatenation separated by an interleaver were proposed in [Pyndiah *et al*.-94] with linear block codes (BCH codes for instance) or in [Benedetto *et al*.-98] with convolutional codes. The serial concatenation of linear block codes had actually already been proposed by Elias in [Elias-54]. In [Pyndiah *et al*.-94] and [Pyndiah-98], a soft decoding algorithm for BCH codes was proposed, enabling the iterative "turbo" decoding of this code. They were therefore called Block Turbo Codes or Turbo Product Codes (TPC). Serial turbo codes have the advantage of a larger minimum distance than parallel turbo codes, with lower complexity component codes. This higher minimum distance avoids the so-called "error floor" that is observed with parallel turbo codes with RSC encoders of reasonable complexity ($K = 4$ for instance) (see Figure 1-10). This means higher coding gain at very low bit error rates (below 10^{-6}). The disadvantage of serial concatenations of convolutional codes is that they converge at higher signal to noise ratios. For instance, within the MOHMS project (Modem for Higher Order Modulation Schemes) funded by ESA [Benedetto *et al*.-05], serial concatenation shows a degradation over parallel concatenation ranging from 0.1 dB for long frame sizes up to 0.5 dB for small frame sizes. To give a complete picture of the state-of-the art codes that can approach the Shannon bound by less than 1 dB, we also mention the class of LDPC codes, proposed by Gallager [Gallager-63] and rediscovered by MacKay [MacKay *et al*.-96], which achieve performance comparable to turbo codes. All these powerful codes are decoded iteratively, using the iterative (turbo) decoding principles of information exchange.

Parallel concatenations of RSC encoders are considered in this manuscript. Yet, most of the techniques presented here are generally applicable to serial concatenations of convolution codes as well.

Example 1-4:

Let us consider the two double-binary CRSC codes (3,2,3) of rate 2/3 of Figure 1-9 that are concatenated in parallel in order to build a rate-1/2 parallel turbo code. Since the CRSC component encoders take 2 input bits, the corresponding turbo code is called double-binary turbo code. Moreover the vector $\mathbf{d}_k$ is called a double-binary symbol.

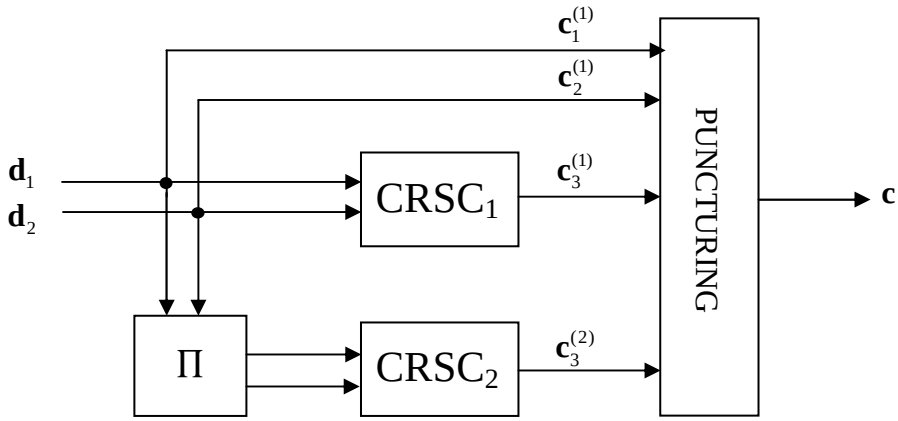


Figure 1-9 : Turbo code constructed as a parallel concatenation of two CRSC code of rate 2/3, with a symbol interleaver

The information vector **d,** also called "information frame", is fed into the first encoder RSC_1 producing the encoded vector $\mathbf{c}^{(1)}$. The information vector is then interleaved according to permutation Π. An interleaver operating on symbols (*i.e.* bits of symbol $\mathbf{d}_k$ remain in the same symbol before and after interleaving) has been proposed by Berrou in [DVB-RCS]. This solution enables a symbol decoder resulting in improved performance [Douillard *et al*.-05]. The interleaved vector is then encoded by a second encoder RSC_2 producing the encoded vector $\mathbf{c}^{(2)}$. The turbo encoded vector **c** includes the systematic bits of RSC_1, and the parity bits generated by RSC_1 and RSC_2. The systematic bits of RSC_2 are not included, since they correspond to the same bits as those of RSC_1 in a different order. Hence, the encoded vector at time *t* is given by:

$$\mathbf{c}_k = \left(c_{k,1}^{(1)}, c_{k,2}^{(1)}, c_{k,3}^{(1)}, c_{k,3}^{(2)}\right)^T = \left(d_{k,1}, d_{k,2}, c_{k,3}^{(1)}, c_{k,3}^{(2)}\right)^T \quad \textbf{(1-18)}$$

In order to generate the desired code rate, puncturing may be applied on the above-produced "mother-codeword". In particular, the redundancy bits are often punctured to obtain the desired code rate.

1.3.3 Interleaver design

1.3.3.1 Overview

The role of the interleaver is very critical and it is therefore a key component of a turbo code, which greatly impacts its performance. The length of the interleaver plays an important role, since the longer the interleaver is, the better the performance [Dolinar *et al.*-98]. This general statement is based on the theory of Shannon. For a fixed length, several interleaver design methodologies have been suggested so far in the literature. These design methodologies aim to fulfil two performance requirements.

The first performance requirement of the code is to converge at low signal to noise ratios (SNRs). In the convergence region, the bit error rate drops significantly after several decoding iterations. This region of the error performance curve is called the "waterfall" region (see Figure 1-10) and is characterized by the interleaver gain, which corresponds to the gain in dB over the uncoded transmission performance curve. The convergence properties are mainly influenced by correlation properties between the two decoding dimensions, the correlation properties of a turbo code being largely influenced by the interleaver design, as will be shown in chapter 3.

The second requirement concerns the asymptotic performance of the code at high SNR. At high SNR, the performance of iterative decoding is close to maximum likelihood decoding. But the turbo coding scheme usually suffers from a few codewords of low Hamming distance. This part of the performance curve is called the "error floor" since the improvement for increasing SNR is very small (see Figure 1-10). It has been shown in [Benedetto *et al.*-96a] and [Perez *et al.*-96] that the asymptotic performance depends on the distribution of the low-weight codewords. In this region, the performance curve of the turbo code is parallel to the performance curve of the uncoded transmission. Assuming maximum-likelihood decoding (ML), the frame error rate (FER) is upperbounded by the union bound:

$$FER \le \frac{1}{2} \sum_{w \ge d_{\min}} n(w) erfc\left(\sqrt{w \cdot R \cdot \frac{E_b}{N_0}} \right), \qquad \textbf{(1-19)}$$

where *erfc* is the complementary error function defined as

$$erfc = \frac{2}{\sqrt{\pi}} \int_x^{\infty} \exp(-t^2) dt \,, \qquad \textbf{(1-20)}$$

R is the code rate, $d_{\min}$ is the minimum distance of the code, $n(w)$ is the number of codewords with weight w, and E_b/N_0 is the signal to noise ratio. The asymptotic performance is then characterized by its asymptotic gain, which corresponds to the gain over the uncoded transmission.

A low-weight codeword for the turbo code occurs when the two RSC codes produce a low-weight codeword. The goal of the interleaver, if low-weight codewords are to be avoided, is to ensure that an information sequence generating a low-weight codeword for one of the RSC encoders is interleaved in such a way as to produce a high-weight codeword for the second RSC encoder.

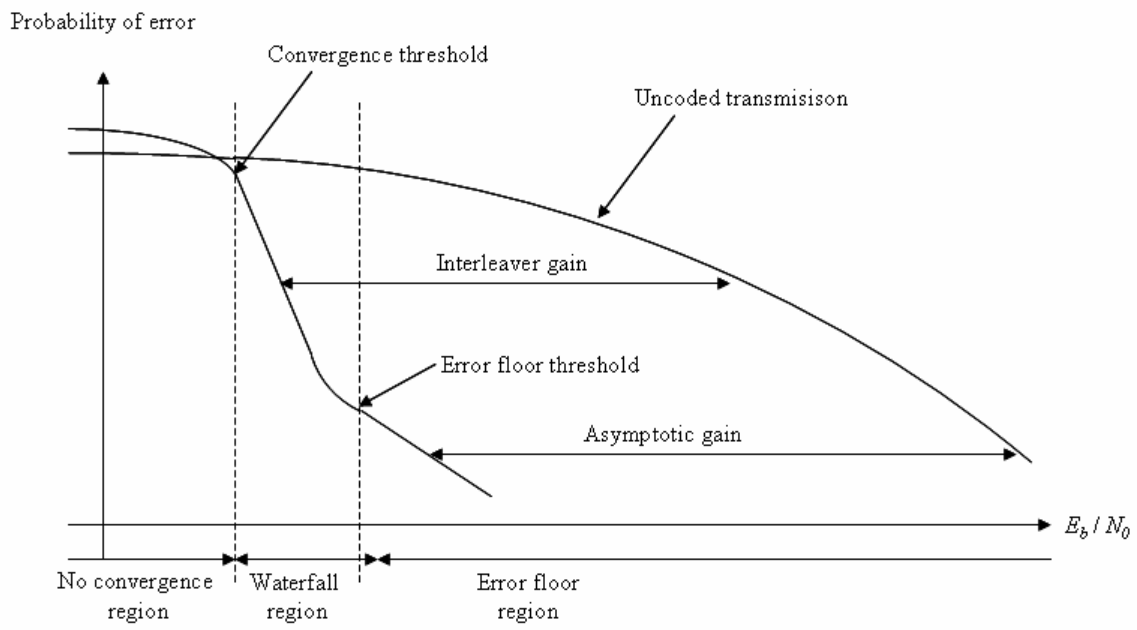


Figure 1-10: Description of the regions of the performance curve of a turbo code

1.3.3.2 Random and pseudo-random interleavers

Random interleavers can be used to satisfy the convergence criterion. This kind of interleaver performs well on average. Unfortunately, the probability of finding a random interleaver, for which no couple of symbols exists that remain close to each other before and after interleaving, is close to 0. Hence, the performance of the code is improved thanks to constraints imposed on the interleaver.

For this reason, S-random interleavers were introduced in [Dolinar *et al*.-95]. This random interleaver construction introduces a constraint in the choice of the permuted indices. At each step of the incremental construction, a random permuted index is selected, which verifies the so-called *spread* condition: two symbols are at least *S* symbols distant in either the natural or the interleaved order. It was shown that for a choice of *S* around $\sqrt{N/2}$, this incremental random construction algorithm succeeds in a reasonable lenght of time.

The spread constraint considers only the relation between each pair of two symbols in the choice of pseudo-random interleavers. Yet the performance of the code with iterative decoding is also influenced by the relations between more than two symbols of the frame. To this end, it was shown in [Hokfelt *et al*.-99] that it depends on the correlation of the internal values of the decoder, which is related to the choice of the interleaver. Using the design technique proposed in [Hokfelt *et al*.-99], pseudo-random interleavers can be designed under a constraint of minimal correlation.

Unfortunately, for pseudo-random permutations, the “error floor” of a two-dimensional turbo code does not increase with the size of the frame [Kahale *et al*.-98], as one would expect. A lower “error floor” can be generally achieved by improving the interleaver construction, instead of using pseudo-random interleavers.

1.3.3.3 Structured interleavers

Using a structured interleaver design matched to the component codes makes it possible to analyse low-weight codewords [Dolinar *et al*.-95],[Kahale *et al*.-98] and [Crozier *et al*.-02]. Hence the objective of the interleaver design is to find a class of structured interleavers, which avoid most of the low-weight codewords. The most promising class of interleaver structures are Almost Regular Permutation (ARP) interleavers [Berrou *et al*.-04] (already introduced in 2000 in [DVB-RCS]) and Dithered Relative Prime (DRP) [Crozier *et al*.-01] interleavers, which are in fact equivalent.

The choice of a specific interleaver among a class of structured interleavers is made so that it has the best distribution of low-weight codewords, and as a consequence, the best asymptotic performance [Crozier *et al*.-03][Berrou *et al*.-04]. Algorithms such as the one proposed in [Garello *et al*.-01], which yields the exact evaluation of the distribution of low-weight codewords, are expensive in computational power, however. As a result, it is difficult to compare a large number of interleavers in order to select the best one. Low complexity algorithms based on the iterative decoder have been introduced to tackle this problem. The "error impulse method" proposed by Berrou *et al.* in [Berrou *et al*.-02] makes it possible to rapidly evaluate the minimum distance of the code. More recently, a similar approach called the "all zero iterative decoding algorithm" has been developed by Garello *et al* [Garello *et al*.-04] and yields the first lower terms of the codeword weight distribution. Nevertheless, structured interleavers, because of their higher regularity, might converge at higher SNR than random interleavers.

1.3.3.4 Example of structured interleaver

ARP interleavers represent a good compromise between structure and irregularity. Let *k'* and *k* denote the indices of the symbols in natural and interleaved order, respectively. The symbol with index *k* in the interleaved order corresponds to the symbol in the natural order with index $k' = \Pi_S(k)$ given by (1-21). Then, the interleaved index for an ARP interleaver is given by:

$$\Pi_S(k) = \lambda \cdot k + \mu(k \bmod 4) + 1 \quad [N] \qquad \textbf{(1-21)}$$

- λ is an integer relatively prime with N chosen in order to guarantee a minimal spreading of the symbols between the natural and interleaved order.
- μ (k mod 4) is an integer offset. The role of parameters μ is to introduce a local disorder in the permutation, and thus to add more irregularity in the interleaver. They are chosen in order to maximize the asymptotic performance of the code.

This type of interleaver can be used for the turbo code presented in the previous section. An additional local disorder is performed by adding an intra-symbol permutation. For even indices *j*, the input couple $\left(d_{k,1}, d_{k,2}\right)$ is permuted into the couple $\left(d_{k,2}, d_{k,1}\right)$. For the DVB-RCS turbo-code [Douillard *et al*-00][DVB-RCS], $\mu(0) = 0$, $\mu(1) = N/2 + P_1$, $\mu(2) = P_2$, $\mu(3) = N/2 + P_3$. The coefficients λ, P_1, P_2, P_3 P_3 are optimized using a "try and select best" search algorithm. They can be optimized independently for each frame size N and each code rate.

Besides, P_1, P_2, P_3 have to be selected in order to ensure the bijection property of the interleaver Π_S. A straightforward way to satisfy this necessary condition is to choose all μ multiples of 4.

It is worth noting that this interleaver has a closed-form expression, meaning that the interleaved addresses can be computed on the fly by dedicated logic. This solution represents a important advantage over random and pseudo-random interleavers, for which a memory is required to store the interleaved addresses.

In this sub-section, we have focused on the design of interleavers in order to maximize the performance of the code. However, as will be addressed in the next chapter, the interleaver induces the main constraints on a practical implementation of the corresponding iterative decoder.

1.4 Turbo decoding

After a brief overview of the turbo decoding principle introducing the concept of Soft-Input Soft-Output (SISO) we will derive the fundamental relation of iterative decoding. This fundamental relation will then be used to describe the principle of iterative turbo decoding.

1.4.1 Overview

Exhaustive MAP decoding of turbo codes is inapplicable because of the presence of the interleaver which would lead to a trellis of a huge number of states, which cannot be considered in practice. For this reason, along with a turbo code construction proposal, a sub-optimal but powerful decoding technique of lower complexity was proposed by Berrou *et al.* in [Berrou *et al*.-93a]: each component code is decoded separately by an appropriate decoder. Each decoder produces information related to its associated encoder. But, since these component codes correspond to the same information message encoded in different orders, the information produced by the component decoders can be combined: the information produced by one decoder is used by the other. This exchange of information between the two component decoders is the basis of the iterative decoding principle. The iterative decoding principle enables the reliability of the decision to be improved during the iterations thanks to an exchange of the so-called *extrinsic information*.

Let us now describe the principle of the SISO algorithm based on the MAP criterion and in particular the concept of extrinsic information. This algorithm will be used for iterative decoding of turbo codes.

1.4.2 Fundamental relation of iterative decoding

In this section, we propose an analytical expression of the expression fundamental relation of iterative decoding for *m*-binary RSC codes. In particular, we introduce the computation of extrinsic information. For a more general expression for non-systematic convolutional codes, we refer the interested reader to [Benedetto et al.-96b].

Let $\mathbf{d} \equiv \mathbf{d}_0^{N-1} = (\mathbf{d}_0 \cdots \mathbf{d}_k \cdots \mathbf{d}_{N-1})$ denote the information vector, where $\mathbf{d}_k = (d_{k,1} \cdots d_{k,l} \cdots d_{k,m})^T$ is also denoted by the scalar quantity $\delta_k = \sum_{l=1}^{m} 2^{l-1} d_{k,l}$, taking values between 0 and $2^m - 1$, and is then written as $\mathbf{d}_k \equiv \delta_k$. Let $\mathbf{c}$ be the encoded sequence $\mathbf{c} \equiv \mathbf{c}_0^{N-1} = (\mathbf{c}_0 \cdots \mathbf{c}_k \cdots \mathbf{c}_{N-1})$, which is a codeword of C. The k^{th} component of $\mathbf{c}$ is an n-component column vector, $\mathbf{c}_k = (c_{k,1} \cdots c_{k,l} \cdots c_{k,n})^T$, where $c_{k,l} \in \{0,1\}$ for $l = 1, \ldots, n$. For $l \le m$, symbol $c_{k,l}$ represents a systematic bit of the encoded sequence; otherwise it represents a redundancy bit. The encoded sequence is then modulated using a BPSK modulation, where the bits $c_{k,l} \in \{0,1\}$ for $l = 1, \ldots, n$ are mapped to symbols $u_{k,l} \in \{-1, +1\}$. The corresponding modulated sequence is denoted $\mathbf{u} \equiv \mathbf{u}_0^{N-1} = (\mathbf{u}_0 \cdots \mathbf{u}_k \cdots \mathbf{u}_{N-1})$, where $\mathbf{u}_k = (u_{k,1} \cdots u_{k,l} \cdots u_{k,n})^T$. After transmission over the AWGN channel of variance σ^2, the sequence observed at the decoder input is $\mathbf{v} \equiv \mathbf{v}_0^{N-1} = (\mathbf{v}_0 \cdots \mathbf{v}_k \cdots \mathbf{v}_{N-1})$, with $\mathbf{v}_k = (v_{k,1} \cdots v_{k,l} \cdots v_{k,n})^T$. Let denote $\mathbf{u}_{k,l \le m}$ and $\mathbf{v}_{k,l \le m}$ the vectors restricted to their systematic components. Similarly $\mathbf{u}_{k,l > m}$ and $\mathbf{v}_{k,l > m}$ correspond to the restriction to the redundancy components.

Applying the MAP criterion to the symbol $\mathbf{d}_k$ requires computing 2^m *a posteriori* probabilities (APPs), $\Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v})$, where $\delta = 0, \ldots, 2^m - 1$ is one of the 2^m possible values of symbol $\mathbf{d}_k$. The hard estimate, $\hat{\mathbf{d}}_k$, is the binary representation of the value δ providing the greatest APP.

$$\hat{\mathbf{d}}_k = \arg\max_{\delta} \left(\Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v}) \right) \qquad \textbf{(1-22)}$$

Let us define $P_k(\delta) = \Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v})$ as the APPs of symbol δ, $P_k^a(\delta) = p(\mathbf{d}_k \equiv \delta)$ as the *a priori* probability of symbol δ, and $P_k^s(\delta) = p\left(\mathbf{v}_{k,l \le m} \,|\, \mathbf{d}_k \equiv \delta\right)$ as the probability corresponding to the observation on the systematic bits at the output of the channel given $\mathbf{d}_k \equiv \delta$. And let $\mathbf{P}_k = \left(P_k(\delta_0), P_k(\delta_1), ..., P_k(\delta_{2^m-1})\right)^T$, $\mathbf{P}_k^a = \left(P_k^a(\delta_0), P_k^a(\delta_1), ..., P_k^a(\delta_{2^m-1})\right)^T$ and $\mathbf{P}_k^s = \left(P_k^s(\delta_0), P_k^s(\delta_1), ..., P_k^s(\delta_{2^m-1})\right)^T$, respectively, denote the corresponding probability vectors. Then, the fundamental relation of iterative decoding is expressed as

$$P_k(\delta) \propto P_k^s(\delta) \times P_k^a(\delta) \times P_k^e(\delta), \qquad \textbf{(1-23)}$$

where $P_k^e(\delta)$ corresponds to the so-called extrinsic probability for symbol δ [Berrou *et al*-93].

Relation (1-23) indicates that the APP $P_k(\delta)$ for the symbol $\mathbf{d}_k$ is computed as the product of the probability of the observation performed on the symbol δ, the *a priori* probability $P_k^a(\delta)$ of the transmitted symbol to be equal to δ, and the extrinsic probability $P_k^e(\delta)$

computed by using the information about the other symbols. This concept of extrinsic probability, which uses the underlying structure of the component code, is very important in the concept of iterative decoding, as will be shown in the next section.

Let $\mathbf{P}_k^e = \left(P_k^e(\delta_0), P_k^e(\delta_1), ..., P_k^e(\delta_{2^m-1})\right)^T$ denote the vector of the 2^m extrinsic probabilities $P_k^e(\delta)$. Then, (1-23) can be expressed for the 2^m-components vectors as

$$\mathbf{P}_k \propto \mathbf{P}_k^s \times \mathbf{P}_k^a \times \mathbf{P}_k^e . \qquad \textbf{(1-24)}$$

This relation shows from the computation of the APP, that the extrinsic probability can be obtained by suppressing the influence of the SISO inputs, namely the *a priori* probability and the probability of observation of the systematic symbols.

Let us denote $P_k^r(\delta) = p\left(\mathbf{v}_{k,l>m} \middle| \mathbf{u}_k, \mathbf{d}_k \equiv \delta\right)$ the probability corresponding to the observation on the redundancy symbols $\mathbf{v}_{k,l>m}$ given symbol $\mathbf{d}_k \equiv \delta$ and $\mathbf{P}_k^r = \left(P_k^r(\delta_0), P_k^r(\delta_1), ..., P_k^r(\delta_{2^m-1})\right)^T$ the corresponding probability vector. Let also $\mathbf{P} = \left(\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{N-1}\right)^T$, $\mathbf{P}^a = \left(\mathbf{P}_0^a, \mathbf{P}_1^a, \ldots, \mathbf{P}_{N-1}^a\right)^T$, $\mathbf{P}^s = \left(\mathbf{P}_0^s, \mathbf{P}_1^s, \ldots, \mathbf{P}_{N-1}^s\right)^T$, $\mathbf{P}^r = \left(\mathbf{P}_0^r, \mathbf{P}_1^r, \ldots, \mathbf{P}_{N-1}^r\right)^T$ and $\mathbf{P}^e = \left(\mathbf{P}_0^e, \mathbf{P}_1^e, \ldots, \mathbf{P}_{N-1}^e\right)^T$ also denote the vectors including the probabilities from time $k = 0$ to N-1. The description of a SISO decoder used for the iterative decoding of a systematic convolutional code is then depicted in Figure 1-11.

Figure 1-11: Description of the SISO decoder for systematic convolutional codes

The next section will apply the fundamental relation (1-24) to decode iteratively a two-dimensional turbo code using two component SISO decoders.

1.4.3 Iterative decoding of turbo codes

The iterative decoding of two-dimensional turbo codes involves an exchange of information between the two component decoders. This exchange is enabled by linking the *a priori* probability input of one component decoder to the extrinsic probability output provided by the other component decoder (see Figure 1-12). Only the extrinsic information is transmitted to the other decoder, in order to avoid the information produced by one decoder being fed back to itself. This constitutes the basic principle of iterative decoding. Extrinsic probability outputs are interleaved or deinterleaved before the transmission to the *a priori* probability inputs.

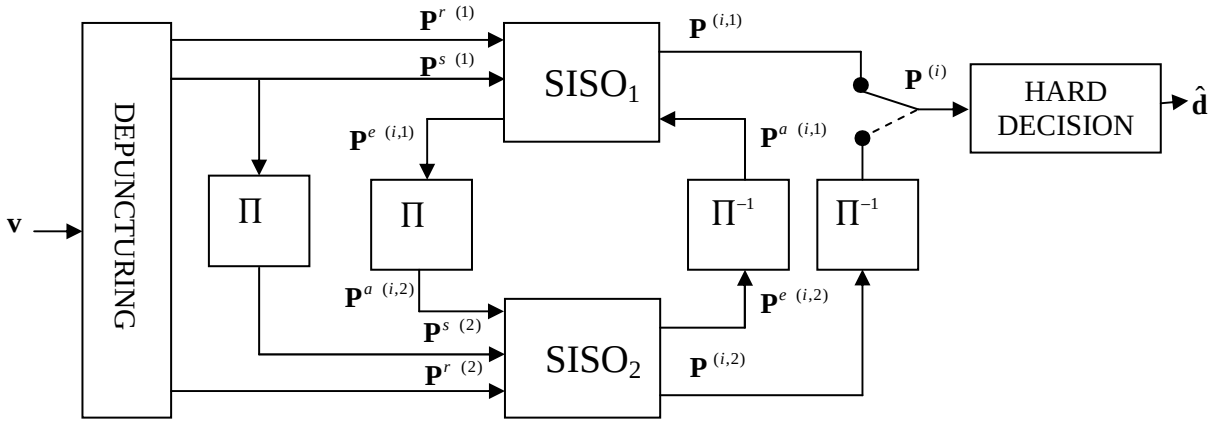


Figure 1-12: Iterative decoder for the parallel concatenation of RSC codes described in Figure 1-8

Let $\mathbf{P}_k^{s\ (1)}$ and $\mathbf{P}_k^{s\ (2)}$ denote the probabilities corresponding to the observation on the systematic symbols for $SISO_1$ and $SISO_2$, respectively. A similar notation is used for the observations made on the redundant symbols. The *a posterior*i, *a priori* and extrinsic probabilities of $SISO_1$ at iteration I are denoted $\mathbf{P}_k^{(i,1)}$, $\mathbf{P}_k^{a\ (i,1)}$ and $\mathbf{P}_k^{e\ (i,1)}$, respectively. Similar notations are used for $SISO_2$.

Let us assume that the natural order ($SISO_1$) and the interleaved order ($SISO_2$) are decoded successively, which constitutes one decoding iteration. At iteration i, the first component decoder uses the observations on the channel $\mathbf{P}_k^{s\ (1)}$ and the *a priori* probability vector $\mathbf{P}_k^{a\ (i,1)}$ provided by the other component decoder at the previous iteration to produce its soft output vectors $\mathbf{P}_k^{e\ (i,1)}$ and $\mathbf{P}_k^{(i,1)}$ according to $\mathbf{P}_k^{(i,1)} \propto \mathbf{P}_k^{s\ (1)} \times \mathbf{P}_k^{a\ (i,1)} \times \mathbf{P}_k^{e\ (i,1)}$.

At the first iteration, since no *a priori* probability is provided by the second component decoder, $\mathbf{P}_k^{a\ (i,1)}$ is initialized to a uniform distribution for the 2^m symbols. The extrinsic information $\mathbf{P}_k^{e\ (i,1)}$ is then interleaved and provided to the other decoder as $\mathbf{P}_{k'}^{a\ (i,2)}$ using $\mathbf{P}_{k'}^{a\ (i,2)} = \Pi\left(\mathbf{P}_k^{e\ (i,1)}\right)$. The second decoder uses in addition its corresponding channel observation $\mathbf{P}_k^{s\ (2)}$ to produce its soft outputs $\mathbf{P}_k^{e\ (i,2)}$ and $\mathbf{P}_k^{(i,2)}$ according to $\mathbf{P}_k^{(i,2)} \propto \mathbf{P}_k^{s\ (2)} \times \mathbf{P}_k^{a\ (i,2)} \times \mathbf{P}_k^{e\ (i,2)}$. The extrinsic probability $\mathbf{P}_k^{e\ (i,2)}$ is then deinterleaved to yield the *a priori* probability of the first decoder at the next iteration (i+1) using $\mathbf{P}_{k'}^{a\ (i+1,1)} = \Pi^{-1}\left(\mathbf{P}_k^{e\ (i,2)}\right)$. After several half-iterations, the iterative process has (hopefully) converged and the hard decision for $\mathbf{d}_k$ is taken using $\mathbf{P}_k^{(i)}$, which corresponds either to $\mathbf{P}_k^{(i,1)}$ or $\Pi^{-1}\left(\mathbf{P}_{k'}^{(i,2)}\right)$. The hard decision symbol $\hat{\delta}$ corresponds to the symbol whose probability $\mathbf{P}_k^{(i)}(\delta)$ is maximal.

The extrinsic information of (1-24) is computed using the MAP criterion. This soft value used for iterative decoding can be computed by a SISO decoder using another algorithm. Other algorithms for m-binary convolution codes, which are not necessarily based on the MAP criterion, are presented in the next section.

1.5 Soft-In Soft-Out decoding based on the BCJR algorithm

In the first step, we outline the SISO decoding algorithms for *m*-binary RSC codes that can be used to provide the soft outputs. These algorithms are compared for double-binary RSC codes. The expression of the most promising algorithm, the BCJR algorithm, is first expressed in the probability domain. Its simplified derivation in the logarithmic domain leads to the Max-Log-MAP algorithm. Finally, we evaluate the complexity of this algorithm and discuss its scheduling.

1.5.1 Overview of SISO decoding algorithms

Soft outputs used in the iterative decoding scheme can be provided by several different SISO algorithms. In order to provide the soft decoding of a RSC code (ν,m,n) three alternative trellis based algorithms are considered:

- the Soft Output Viterbi Algorithm (SOVA). This is a soft output version of the popular Viterbi algorithm (VA) [Forney-73], which produces hard decision bits based on the estimation of the Maximum Likelihood path in the trellis corresponding to the convolutional encoder. The SOVA proposed by Battail in [Battail-87] and Hagenauer and Hoeher in [Hagenhauer *et al*.-89] is a modified VA that produces a soft output by weighting two concurrent paths associated with the hard decision. The first version of this algorithm led to high complexity hardware implementations. Along with the introduction of turbo codes, Berrou *et al.* [Berrou *et al*.-93b] proposed a simplified weighting algorithm, which enabled practical implementation of binary turbo decoders [CAS-95] [Jézéquel *et al*.-97].

- the *Bahl-Cock-Jelinek-Raviv* (BCJR) algorithm [Bahl *et al*.-74], also known as the symbol-by-symbol *Maximum A Posteriori* (MAP) algorithm or as the Forward-Backward algorithm. This trellis-based algorithm computes for each symbol an APP $p(\mathbf{d}_k \equiv \delta \left| \mathbf{v} \right.)$ by evaluating the probabilities of all possible paths from the initial state to the final state in the trellis. The log-domain version of this algorithm allows a reduced complexity implementation by transforming multiplications into sums, without loss of performance. A further (sub-optimal) simplification called the Max-Log-MAP is also often considered [Robertson *et al*.-95]. In this simplified expression, the algorithm involves two Viterbi recursions that are combined to produce the soft outputs [Viterbi-98].

- the Dual-BCJR algorithm. This corresponds to a BCJR algorithm working on the trellis corresponding to the reciprocal dual code of the original convolutional code. The concept of decoding by using the dual code was first addressed by Hartmann and Rudolph in [Hartmann *et al*.-76] and Battail *et al*. in [Battail *et al*.-79]. They found a way to compute the APP $p(\mathbf{d}_k \equiv \delta \left| \mathbf{v} \right.)$ for a binary linear block code using its dual code. For a code C of length n_c and dimension k_c, the number of codewords equals 2^{n_c}. On the other hand, the number of codewords of the dual code is equal

to $2^{n_c-k_c}$. The interest of the concept of decoding by using the dual code is directly related to this reduction in decoding complexity. An expression of this algorithm for tail-biting convolutional codes was developed by Riedel in [Riedel-98] and an additive version working in the logarithmic domain was derived in [Montorsi *et al*.-01b].

The comparison of the SOVA and the Max-Log-MAP algorithm in terms of performance and complexity for turbo codes using binary (m = 1) component codes have been addressed in [Robertson *et al*.-95]. The Max-Log-MAP algorithm which involves two Viterbi recursions is about twice as complex as the SOVA. But, the SOVA has a degradation of 0.2 – 0.4 dB compared to the Max-Log-MAP algorithm. The explanation was given in [Fossorier *et al*.-98]: the SOVA takes into account four paths in the trellis in order to compute its soft output, whereas the Max-Log-MAP algorithm takes into account all possible paths from the initial state to the final state. As a result, the soft outputs produced by the SOVA are less reliable than those of the Max-Log-MAP algorithm.

For turbo codes using double-binary (m = 2) component codes, it was observed in [Saouter *et al*.-02a] that the complexity of the SOVA is about three times the complexity of the Viterbi decoder. An implementation of the Max-Log-MAP algorithm achieving the same throughput requires two Viterbi recursions (a forward and a backward) and the computation of the soft output, whose complexity is equivalent to a Viterbi recursion. Hence the complexity of the Max-Log-MAP algorithm is also about three times the complexity of the Viterbi decoder [Saouter *et al*.-02b], and is therefore equivalent to the SOVA. Both algorithms make use of a symbol decoding technique: for double-binary turbo codes, the extrinsic information is produced for each symbol, instead of for each bit. This symbol decoding technique is made possible by the interleaver, which conserves the grouping of the bits into symbols. Symbol decoding yields 0.1-0.3 dB improvement compared to bit decoding [Sawaya *et al*.-03]. For double-binary turbo codes the degradation of the SOVA compared to the Max-Log-MAP algorithm amounts to 0.1-0.2 dB, for an equivalent complexity. This justifies the use of the Max-Log-MAP in the first practical implementation of a double-binary turbo code decoder [TC1000].

Another alternative for decoding double-binary turbo codes lies in using the reciprocal dual code. In fact, on the one hand the rate of the double-binary RSC code is 2/3 and the associated trellis has 4 transitions per encoder state. On the other hand, the dual code has a rate 1/3 and the associated trellis has 2 transitions per state. This explains the interest of the dual decoding method for the double-binary RSC code. We implemented this algorithm and compared it to the Max-Log-MAP algorithm in terms of performance and complexity. The results of our work is not presented here, but it pointed out several disadvantages of the algorithm working in the dual domain over the classical Max-Log-MAP algorithm:

- no symbol decoding is available since the equations of the BCJR algorithm in the dual domain given in [Riedel-98] produce extrinsic information for each

information bit. The use of a bit decoding technique leads to a performance degradation of 0.2-0.6 dB for double-binary turbo codes [Saouter *et al*.-03].

- the additive expression of the algorithm in the logarithmic domain uses the classical Jacobian logarithm correction [Robertson *et al*.-97]

$$\log(e^x + e^y) = \max(x, y) + \log\left(1 + e^{-|x-y|}\right), \quad \textbf{(1-25)}$$

but also the following correction function:

$$\log(e^x - e^y) = \max(x, y) + \log\left(1 - e^{-|x-y|}\right). \quad \textbf{(1-26)}$$

The implementation of the correction term in (1-26) which his bounded by ln(2) has been widely addressed in the literature: an Look-Up-Table (LUT) with a small number of values [Erfanian *et al*.-94], or even a single value [Classon *et al*.-00]. This correction term can also be totally ignored as is the case in the Max-Log-MAP algorithm [Robertson *et al*.-95]. Nevertheless, the correction term in (1-26) is unbounded and tends towards minus infinity when x and y are close to each other. Consequently, this correction cannot be discarded and it sweeps away the reduction of complexity promised by this algorithm: a rough complexity comparison shows that the two algorithms are equivalent in terms of complexity.

- numerical precision problems may arise for the implementation of the correction term in (1-26). This may considerably increase the number of bits required for coding internal values of this algorithm and thus the complexity of the algorithm.

Among these three disadvantages, the lack of a symbol decoding technique is believed to be the most important. In fact, the other two disadvantages: mandatory look-up tables and numerical precision problems, could be overcomed as has been done for the decoding of LDPC codes in [Zhang *et al*.-01] and [Blanksby *et al*.-02], where the function $\log\tanh(x) = \log(e^{2x} - 1) - \log(e^{2x} + 1)$ has been implemented with a look-up table.

In our case of double-binary turbo codes, the search for a symbol decoding technique for the Dual-BCJR algorithm has been investigated at ENST Bretagne, recently, but it has been unsuccessful so far. For this reason, the Max-Log-MAP algorithm has been used in our work for improving the throughput of the turbo decoder for double-binary turbo codes. The derivation of this algorithm is described hereafter for the general case of *m*-binary turbo codes.

1.5.2 BCJR MAP decoding

As has been described above, the BCJR algorithm operates on the trellis representing a Markov process, such as an RSC encoder. It has been stated that a symbol decoding can be used for this algorithm, which involves the computation of 2^m APPs:

$$\Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v}) = \frac{p(\mathbf{d}_k \equiv \delta, \mathbf{v})}{p(\mathbf{v})} = \frac{p(\mathbf{d}_k \equiv \delta, \mathbf{v})}{\sum_{\delta'=0}^{2^m-1} p(\mathbf{d}_k \equiv \delta', \mathbf{v})}. \quad \textbf{(1-27)}$$

In practice, we compute the 2^m joint likelihoods $p(\mathbf{d}_k \equiv \delta, \mathbf{v})$, the APPs being inferred with a simple normalization operation according to (1-27).

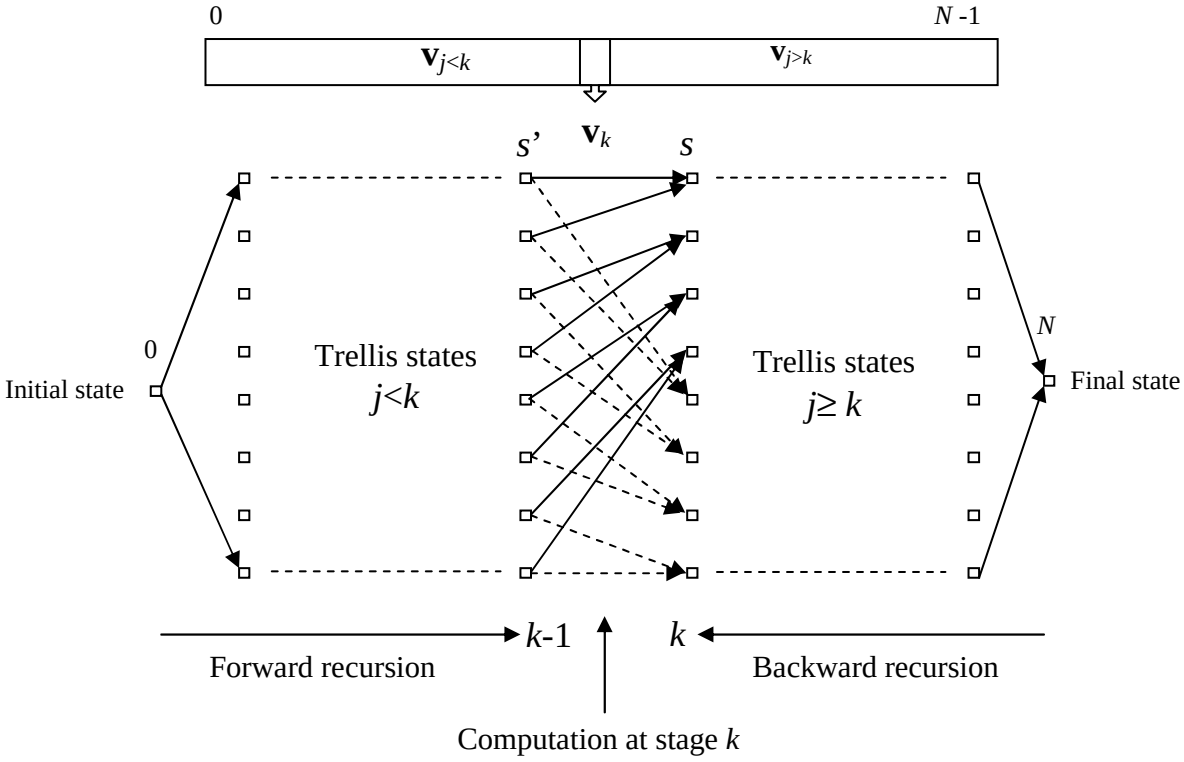


Figure 1-13: Computations at time *k* of the BCJR algorithm on the trellis of the code

The BCJR algorithm computes the probabilities of transitions $p(s_k = s', s_{k+1} = s, \mathbf{v})$ between two states *s'* and *s* of the trellis. The joint likelihoods are then obtained by considering all possible transitions (*s'*,*s*) for which $\mathbf{d}_k \equiv \delta$:

$$p(\mathbf{d}_k \equiv \delta, \mathbf{v}) = \sum_{(s',s)\in\Gamma_\delta} p(s_k = s', s_{k+1} = s, \mathbf{v}), \qquad \textbf{(1-28)}$$

where Γ_δ represents the set of state transitions (*s'*,*s*) in the trellis caused by the transmission of the *m*-binary symbol data $\mathbf{d}_k \equiv \delta$. Assuming a memoryless channel, the components of the observation vector **v** are independent, and the probabilities of transitions $p(s_k = s', s_{k+1} = s, \mathbf{v})$ are split into three parts (see Figure 1-13) according to equation (1-29):

- the "past" depending on state *s'* and on observations for symbols with indices $j < k$.
- the "present" gathering observations on the symbol with index $j = k$.
- the "future", depending on state *s* and on observations for symbols with indices $j > k$.

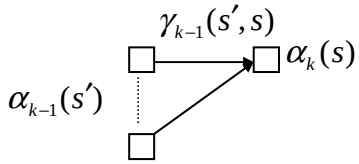
$$p(s_k = s', s_{k+1} = s, \mathbf{v}) = \underbrace{p(s_k = s', \mathbf{v}_{k<j})}_{\text{"past"}} \times \underbrace{p(s_{k+1} = s, \mathbf{v}_k \mid s_k = s')}_{\text{"present"}} \times \underbrace{p(\mathbf{v}_{k>j} \mid s_{k+1} = s)}_{\text{"future"}} \qquad \textbf{(1-29)}$$

Using (1-27) and (1-29), the APPs are computed as:

$$\Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v}) = \frac{\sum_{(s',s)\in\Gamma_\delta} \alpha_k(s')\gamma_k(s',s)\beta_{k+1}(s)}{\sum_{(s',s)} \alpha_k(s')\gamma_k(s',s)\beta_{k+1}(s)} \qquad \textbf{(1-30)}$$

where

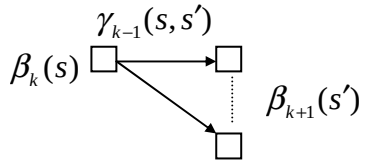
- $\alpha_k(s') = p(s_k = s', \mathbf{v}_{k<j})$ is the forward state probability of states $s_k = s'$, which is computed recursively as



$$\alpha_k(s) = \sum_{s'\in\Gamma_s^-} \alpha_{k-1}(s')\gamma_{k-1}(s',s) \text{ for } k = 1,\ldots,N \qquad \textbf{(1-31)}$$

where Γ_s^- is the set of preceding states of *s*.

- $\beta_{k+1}(s) = p(\mathbf{v}_{k>j} \,|\, s_{k+1} = s)$ is the backward state probability of state $s_{k+1} = s$, which is computed recursively as



$$\beta_k(s) = \sum_{s'\in\Gamma_s^+} \beta_{k+1}(s')\gamma_k(s,s') \text{ for } k = N-1,\ldots,0 \qquad \textbf{(1-32)}$$

where Γ_s^+ is the set of succeeding states of *s*.

In practice, in order to avoid any precision or overflow problem in the numerical representation of the forward and backward states probabilities, they have to be regularly normalized during the recursive computation (1-31) and (1-30) [Berrou *et al*.-96].

Recursion initialization:

The boundary conditions $\boldsymbol{\alpha}_0$ and $\boldsymbol{\beta}_N$ depend on the knowledge of the encoder state at the beginning and at the end of the encoding process: if s_0 is known, then $\alpha_0(s_0) = 1$ and $\alpha_0(s) = 0$ for every $s \neq s_0$; if s_0 is unknown, $\boldsymbol{\alpha}_0$ is initialized with a uniform probability distribution. The same rule is applied for backward probabilities $\boldsymbol{\beta}_N$ with final state s_N. Recursion initialization for circular codes will be described in section 1.5.6.

Computation of the transition probability:

Let $\gamma_k(s',s) = p(s_{k+1} = s, \mathbf{v}_k \,|\, s_k = s')$ be the transition probability from state $s_k = s'$ to state $s_{k+1} = s$. If no transition exists between states *s'* and *s*, the transition probability equals 0, otherwise it is computed using the Bayes' relation twice:

$$p(s_{k+1} = s, \mathbf{v}_k \,|\, s_k = s') = p(\mathbf{v}_k \,|\, s_k = s', s_{k+1} = s) \times p(s_{k+1} = s \,|\, s_k = s') \qquad \textbf{(1-33)}$$

The first term of the product in (1-33) corresponds to the conditional probability of the channel:

$$p(\mathbf{v}_k \mid s_k = s', s_{k+1} = s) = p(\mathbf{v}_k \mid \mathbf{d}_k) = p(\mathbf{v}_k \mid \mathbf{u}_k) \quad \textbf{(1-34)}$$

The second term in (1-33) corresponds to the probability of a transition from state *s'* to state *s* conditioned to state *s'*, *i.e.* the probability of emission of $\mathbf{d}_k \equiv \delta$, the *a priori* probability of symbol $\mathbf{d}_k$. This *a priori* probability is provided by the other component decoder in the iterative decoding process. When the transition exists, the transition probability can be rewritten using (1-33) as

$$\gamma_k(s', s) = p(\mathbf{v}_k \mid \mathbf{u}_k) \times p(\mathbf{d}_k \equiv \delta). \quad \textbf{(1-35)}$$

Assuming a gaussian memoryless channel, the conditional probability of the channel can be written using (1-6) as:

$$p(\mathbf{v}_k \mid \mathbf{u}_k) = \prod_{l=1}^{n} \left(\frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(v_{k,l} - u_{k,l})^2}{2\sigma^2} \right) \right). \quad \textbf{(1-36)}$$

Developing the $(v_{k,l} - u_{k,l})^2$ and keeping only the terms specific to the transition considered leads to

$$p'(\mathbf{v}_k \mid \mathbf{u}_k) = \exp\left(\frac{\sum_{l=1}^{n} v_{k,l} \cdot u_{k,l}}{\sigma^2} \right). \quad \textbf{(1-37)}$$

This probability is used in practice for computating the probabilities of transitions, since the other terms are constant whatever the transition considered, and thus disappear in the ratio of the computation $p(\mathbf{d}_k \equiv \delta \mid \mathbf{v})$ from (1-30).

As was shown in the previous section, a SISO decoder needs to produce extrinsic information. This is computed by considering the APP information about all the received symbols except the direct information related to the symbol considered. By extracting all the terms containing any direct information about $\mathbf{d}_k$ out of the product of (1-36), the transition probability is computed as

$$\gamma_k(s', s) = \gamma_k^s(\mathbf{d}_k \equiv \delta) \cdot \gamma_k^e(s', s), \quad \textbf{(1-38)}$$

where

$$\gamma_k^e(s', s) = \frac{1}{\sigma\sqrt{2\pi}} \prod_{l=m+1}^{n} \exp\left(-\frac{(v_{k,l} - u_{k,l})^2}{2\sigma^2} \right), \quad \textbf{(1-39)}$$

and

$$\gamma_k^s(\mathbf{d}_k \equiv \delta) = p(\mathbf{d}_k \equiv \delta) \cdot \frac{1}{\sigma\sqrt{2\pi}} \prod_{l=1}^{m} \exp\left(-\frac{(v_{k,l} - u_{k,l})^2}{2\sigma^2} \right). \quad \textbf{(1-40)}$$

Let us define the conditional probability $p(\mathbf{v}_{k,l\le m} \mid \mathbf{d}_k)$ on systematic symbols as:

$$p(\mathbf{v}_{k,l\le m} \mid \mathbf{d}_k) = \frac{1}{\sigma\sqrt{2\pi}} \prod_{l=1}^{m} \exp\left(-\frac{(v_{k,l} - u_{k,l})^2}{2\sigma^2} \right). \quad \textbf{(1-41)}$$

Fundamental relation of iterative decoding:

Putting (1-35) into (1-30), and extracting $\gamma_k^s(\mathbf{d}_k \equiv \delta)$, which does not depend on states (s,s') yields:

$$\Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v}) = p(\mathbf{v}_{k,l\le m} \,|\, \mathbf{d}_k) \times p(\mathbf{d}_k \equiv \delta) \times \frac{\sum_{(s',s)\in\Gamma_\delta} \alpha_k(s')\gamma_k^e(s',s)\beta_{k+1}(s)}{\sum_{(s',s)} \alpha_k(s')\gamma_k^e(s',s)\beta_{k+1}(s)}. \qquad \textbf{(1-42)}$$

This relation corresponds to the fundamental relation of iterative decoding (1-23) given in the previous section. It exhibits the extrinsic probability for symbol δ, which is calculated as:

$$P_k^e(\delta) = \frac{\sum_{(s',s)\in\Gamma_\delta} \alpha_k(s')\gamma_k^e(s',s)\beta_{k+1}(s)}{\sum_{(s',s)} \alpha_k(s')\gamma_k^e(s',s)\beta_{k+1}(s)} \qquad \textbf{(1-43)}$$

1.5.3 Log-MAP decoding

MAP decoding with the BCJR algorithm requires a large number of operations, involving multiplications and exponentiations. This justifies that it is generally implemented in the logarithmic domain, where the multiplications become additions, which is more adapted to hardware implementation.

Definitions:

Let us define the following soft information variables to be proportional to the log of their respective probabilities:

- $L_k(\delta) = -\frac{\sigma^2}{2}\log \Pr(\mathbf{d}_k \equiv \delta \,|\, \mathbf{v})$ as the *a posteriori* log-likelihood
- $L_k^a(\delta) = -\frac{\sigma^2}{2}\log p(\mathbf{d}_k \equiv \delta)$ as the *a priori* log-likelihood

Let us define the forward and backward state metrics as

$$a_k(s) = -\sigma^2 \log \alpha_k(s) \text{ and } b_k(s) = -\sigma^2 \log \beta_k(s) \qquad \textbf{(1-44)}$$

and the branch metrics as

$$c_k(s',s) = -\sigma^2 \log \gamma_k(s',s) \text{ and } c_k^e(s',s) = -\sigma^2 \log \gamma_k^e(s',s) \qquad \textbf{(1-45)}$$

Computation of the branch metrics:

Introducing the (1-45), and the simplified version (1-37) of the transition probability into (1-38) and (1-39) yields

$$c_k(s',s) = -\sigma^2 \log \exp\left(p(\mathbf{d}_k \equiv \delta) + \frac{\sum_{l=1}^{n} v_{k,l} \cdot u_{k,l}}{\sigma^2} \right) = 2L_k^a(\delta) - \sum_{l=1}^{n} v_{k,l} \cdot u_{k,l}\,, \qquad \textbf{(1-46)}$$

and

$$c_k^e(s',s) = -\sigma^2 \log \exp\left(\frac{\sum_{l=m+1}^{n} v_{k,l} \cdot u_{k,l}}{\sigma^2} \right) = -\sum_{l=m+1}^{n} v_{k,l} \cdot u_{k,l}\,. \qquad \textbf{(1-47)}$$

Forward and backward state metrics computations:

Computing the forward and backward state metrics involves the computation of the logarithm of a sum of exponentials

$$-\sigma^2 \log\left(e^{-\frac{x}{\sigma^2}} + e^{-\frac{y}{\sigma^2}} \right), \qquad \textbf{(1-48)}$$

which is computed as the classical Jacobian logarithm correction [Robertson *et al*.-97] as follows:

$$-\sigma^2 \log\left(e^{-\frac{x}{\sigma^2}} + e^{-\frac{y}{\sigma^2}} \right) = \min(x,y) - \sigma^2 \log\left(1 + e^{-\frac{|x-y|}{\sigma^2}} \right) = \min^*(x,y)\,. \qquad \textbf{(1-49)}$$

This function of two operands is denoted *min**. The second term of this expression can be easily implemented with a LUT as shown in Figure 1-14. Moreover, for several operands, this function can be computed recursively thanks to its associative property, using multiple *min** operators of two operands $\min^*(x,y,z) = \min^*\left(\min^*(x,y),z\right)$.

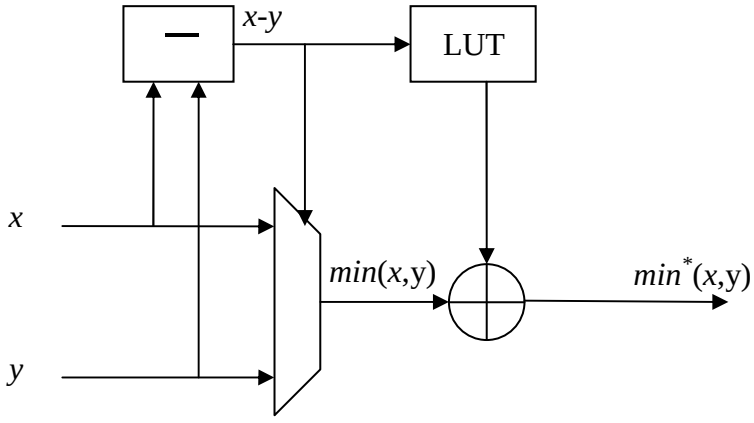


Figure 1-14: Architecture of the *min** **operation using a Look-Up Table**

Adopting another definition for the state metrics in (1-44) by omitting the *minus* sign leads to replacing the *min* operator by the *max* operator in the computation of the logarithm of the

sum of exponential[1]. This explains the name of the algorithm: Max-Log-MAP. The definitions (1-44) and (1-45) have been chosen in order to compute positive metrics and soft outputs. Hence, the computation of the forward and backward state metrics (replacing (1-44) and (1-45) into (1-31) and (1-32)) and using the *min** operator yields:

$$a_k(s) = \min_{s' \in \Gamma_s^-}{}^{*} \left(a_{k-1}(s') + c_{k-1}(s', s) \right) \text{ for } k = 1, \cdots, N \quad \textbf{(1-50)}$$

$$b_k(s) = \min_{s' \in \Gamma_s^+}{}^{*} \left(b_{k+1}(s') + c_k(s, s') \right) \text{ for } k = N-1, \ldots, 0 \quad \textbf{(1-51)}$$

If the initial (final) state of the encoder is known, the forward (respectively backward) recursion is initialized with 0 for the initial starting state and +∞ for the other states. If not, all state metrics are initialized to an identical value, 0 for instance.

Computation of the *a posteriori* likelihood and hard decision:

Let us define the soft information $\lambda_k(\delta)$ of the symbol δ at time *k* as

$$\lambda_k(\delta) = \min_{(s',s) \in \Gamma_\delta}{}^{*} \left(a_k(s') + c_k(s', s) + b_{k+1}(s) \right). \quad \textbf{(1-52)}$$

Then using (1-52) in (1-30), the *a posteriori* log-likelihood can be computed as:

$$L_k(\delta) = \frac{1}{2} \left(\lambda_k(\delta) - \min_{\delta'} \lambda_k(\delta') \right), \quad \textbf{(1-53)}$$

where the subtraction corresponds to the normalization. This normalization makes an approximation by using the *min* operator instead of the *min** operator, which is required by a direct expression of the denominator of (1-30) in the logarithmic domain. But, it has no impact on performance, and makes it possible to have $L_k(\delta)$ equal to 0 when δ corresponds to the hard decision symbol. Actually, the hard decision provided by the decoder corresponds to the binary representation of the symbol δ, $\delta = 0, \ldots, 2^m\text{-}1$ that minimizes $\lambda_k(\delta)$:

$$\hat{\delta} = \arg\min_{\delta} \left(L_k(\delta) \right) = \arg\min_{\delta} \left(\lambda_k(\delta) \right) \quad \textbf{(1-54)}$$

Computation of the extrinsic information:

In a similar manner as for the *a posteriori* likelihood, let us define the extrinsic soft information $\lambda_k^e(\delta)$ at time *k* as

$$\lambda_k^e(\delta) = \min_{(s',s) \in \Gamma_\delta}{}^{*} \left(a_k(s') + c_k^e(s', s) + b_{k+1}(s) \right). \quad \textbf{(1-55)}$$

Because of the approximation performed on the normalization (1-53), the normalization of extrinsic information computed from (1-43) is modified in order to write a relation similar to (1-42) in the logarithmic domain. Hence, extrinsic information is normalized by subtracting the extrinsic value corresponding to the hard decision $\hat{\delta}$, which nullifies $L_k(\delta)$.

$$L_k^e(\delta) = \frac{1}{2} \left(\lambda_k^e(\delta) - \lambda_k^e(\hat{\delta}) \right) \quad \textbf{(1-56)}$$

Then, the fundamental relation of iterative decoding (1-42) can be reformulated in the logarithmic domain as

[1] Note that both algorithms are strictly equivalent in terms of performance.

$$L_k(\delta) = -\frac{1}{2}\sum_{l=1}^{m} v_{k,l} \cdot \left(u_{k,l}\Big|_{\delta_k=\delta} - u_{k,l}\Big|_{\delta_k=\hat{\delta}} \right) + L_k^a(\delta) - L_k^a(\hat{\delta}) + L_k^e(\delta), \quad \textbf{(1-57)}$$

where the notation $u_{k,l}\Big|_{\delta_k=\delta}$ specifies that the *l*-th bit value of symbol $\mathbf{u}_k$ corresponds to information symbol $\delta_k = \delta$. This relation shows that the soft outputs $L_k^e(\delta)$ and $L_k(\delta)$ can be computed from each other by a simple addition (or subtraction). Thanks to the coefficient 1/2 in the definitions of $L_k(\delta)$, $L_k^a(\delta)$ and $L_k^e(\delta)$, the decoder provides soft estimates $L_k(\delta)$ and extrinsic information $L_k^e(\delta)$ on the same scale as noisy sample $v_{k,l}$ [1].

Noise variance:

It is worth noting that this algorithm requires an estimation of the noise variance σ^2 of the Gaussian channel. This noise variance can be provided by an external estimator. Alternatively, the algorithm can work in some cases using a fixed σ^2 corresponding to the operating point of the decoder.

1.5.4 Max-Log-MAP decoding

The Log-MAP algorithm suffers from two drawbacks: the implementation of the correction factor in the *min** function by an LUT leads to an augmentation in complexity and a reduction of the maximal clock frequency (see chapter 5.3). Moreover, for double-binary turbo codes (*m* = 2), it has been shown in [Douillard *et al*.-05] that the degradation associated with the Max-Log-MAP algorithm is below 0.1 dB compared to the optimal log-MAP algorithm.

The Max-Log-MAP algorithm is obtained from the Log-MAP algorithm by replacing the *min** operator by a simple *min* operator. This sub-optimal algorithm corresponds to two Viterbi recursions: one in the forward direction computing state metrics $\mathbf{a}_k$ recursively, and the other in the backward direction computing state metrics $\mathbf{b}_k$ recursively. Hence the decoding equations then summarized as follows:

$$c_k(s',s) = 2L_k^a(\delta) - \sum_{l=1}^{n} v_{k,l} \cdot u_{k,l} \quad \textbf{(1-58)}$$

$$c_k^e(s',s) = -\sum_{l=m+1}^{n} v_{k,l} \cdot u_{k,l} \quad \textbf{(1-59)}$$

$$a_k(s) = \min_{s' \in \Gamma_s^-} \left(a_{k-1}(s') + c_{k-1}(s',s) \right) \text{ for } k = 1,\ldots,N \quad \textbf{(1-60)}$$

$$b_k(s) = \min_{s' \in \Gamma_s^+} \left(b_{k+1}(s') + c_k(s,s') \right) \text{ for } k = N-1,\ldots,0 \quad \textbf{(1-61)}$$

$$\lambda_k(\delta) = \min_{(s',s) \in \Gamma_\delta} \left(a_k(s') + c_k(s',s) + b_{k+1}(s) \right) \quad \textbf{(1-62)}$$

$$\hat{\delta} = \arg\min_{\delta} \left(\lambda_k(\delta) \right) \quad \textbf{(1-63)}$$

[1] The difference $u_{k,l}\Big|_{\delta_k=\delta} - u_{k,l}\Big|_{\delta_k=\hat{\delta}}$ equals 0, -2 or 2, depending on the values of the two terms.

$$L_k(\delta) = \frac{1}{2}\left(\lambda_k(\delta) - \lambda_k(\hat{\delta})\right) \quad \textbf{(1-64)}$$

$$L_k^e(\delta) = \frac{1}{2}\left(\min_{(s',s)\in\Gamma_\delta}\left(a_k(s') + c_k^e(s',s) + b_{k+1}(s)\right) - \min_{(s',s)\in\Gamma_{\hat{\delta}}}\left(a_k(s') + c_k^e(s',s) + b_{k+1}(s)\right)\right) \quad \textbf{(1-65)}$$

$$L_k(\delta) = -\frac{1}{2}\sum_{l=1}^{m} v_{k,l}\cdot\left(u_{k,l}\Big|_{\delta_k=\delta} - u_{k,l}\Big|_{\delta_k=\hat{\delta}}\right) + L_k^a(\delta) - L_k^a(\hat{\delta}) + L_k^e(\delta) \quad \textbf{(1-66)}$$

The degradation in performance of this algorithm is caused by the sub-optimality of the *min* operator, which approximates the *min** operation by an overestimation for all metrics calculations. Hence, the extrinsic information is less reliable than with the Log-MAP algorithm, especially at the beginning of the iterative process and at low signal to noise ratios (*i.e.* when the argument of the min* correction term has a large average value). To cope with the sub-optimality of the extrinsic computation due to the Max-Log approximation, an iteration-dependent scaling of the extrinsic information is often used [Vogt *et al.*-00]. This enables the performance of the decoder to be improved by multiplying the extrinsic information by a coefficient smaller than 1.0. It has been shown in [Chen *et al.*-01] that this coefficient can be computed by the ratio of the average of the extrinsic information produced by the sub-optimal Max-Log-MAP algorithm to the average of the extrinsic information produced by the Log-MAP algorithm. We typically use 0.7 an iteration-independent value, except for the last iteration where extrinsic information is not scaled. Another interest of the Max-Log-MAP algorithm is that it does not require the evaluation of the noise variance σ^2.

1.5.5 Computational complexity of the Max-Log-MAP algorithm

In this section, the computational complexity of the Max-Log-MAP algorithm is evaluated for the RSC code (ν,m,n). The results of this evaluation are summarized in Table 1-2. The computational complexity of an algorithm corresponds to the number of basic operations involved in the algorithm, such as additions, comparison, multiplications, etc.

Computation of the branch metrics:

The trellis of an m-binary RSC code with memory ν is composed of 2^ν states with 2^m branches arriving on each state. There are not $2^\nu \cdot 2^m$ different branch metrics, but only 2^n, since each of the encoded symbol $u_{k,l}$, l = 0, …, n-1, takes a value +1 or -1. The computation of all combinations of $\sum_{l=1}^{n} v_{k,l}\cdot u_{k,l}$ in (1-58) can be computed in a tree-like manner. The optimal number of adders depends on n and is given by $n_{ADD\_BRANCH}(n)$ (see Table 1-2 for values). Thus, the addition of the *a priori* information yields $n_{ADD\_BRANCH}(n) + 2^m$ additions. It also requires 2^m multiplications by the constant 2. In the sequel, multiplications and divisions by a factor of 2 are neglected: using a binary representation of the numbers, multiplications (or divisions) by a power of 2 are implemented with a shift to the left (right, respectively). The computation of the branch metrics is performed twice, once for the forward recursion and once for the backward recursion.

Computation of the state metrics:

The update of one forward state metric according to (1-60) involves the comparison and selection of 2^m concurrent paths that can be straightforwardly implemented using 2^m additions, and 2^m–1 comparison-selection operations, organized in a tree-like manner. The operations involved in the backward recursions (1-61) are similar to those of the forward recursion, and the associated complexity is identical. Extending to the 2^ν states, the computation of one recursion step – forward or backward – requires $2^{\nu+m}$ additions and $2^\nu(2^m-1)$ comparison-selection operations.

Computation of the a posteriori likelihood:

The computation of the *a posteriori* likelihood according to (1-64) requires the computations of the set of 2^m likelihoods $\lambda_k(\delta)$, δ = 0,…,2^m-1 using relation (1-62). It involves two additions per branch. This complexity can be reduced by noting that partial results are already available through the forward or the backward recursion. Indeed, during the forward recursion, the concurrent forward state metrics denoted $a_{k+1}^c(s',s)$, s = 0,…,$2^{\nu-1}$ are computed as

$$a_{k+1}^c(s',s) = a_k(s') + c_k(s',s), \qquad \textbf{(1-67)}$$

which can be used in the computation of (1-62):

$$\lambda_k(\delta) = \min_{(s',s)\in\Gamma_\delta} \left(a_{k+1}^c(s',s) + b_{k+1}(s) \right). \qquad \textbf{(1-68)}$$

An equivalent relation can be written using the concurrent backward state metrics $b_k^c(s',s)$, s' = 0,…,$2^{\nu-1}$:

$$\lambda_k(\delta) = \min_{(s',s)\in\Gamma_\delta} \left(a_k(s') + b_k^c(s',s) \right). \qquad \textbf{(1-69)}$$

The computation (1-66) involves the selection of the minimum among 2^ν concurrent values for $\lambda_k(\delta)$ corresponding to the 2^ν concurrent paths. The comparison is performed in a tree-like manner, resulting in 2^ν–1 comparison-selection operations. Consequently, the computation of the 2^m values for $\lambda_k(\delta)$ requires $2^m 2^\nu$ additions and $2^m(2^\nu-1)$ comparison-selection operations. The hard decision is inferred by selection of the minimum value for $\lambda_k(\delta)$ among the resulting 2^m values, using a comparison-selection tree of 2^m–1 operations. Hence the computation of the 2^m *a posteriori* likelihoods (1-64) requires $2^m(2^\nu+1)-1$ additions and $2^m 2^\nu-1$ comparison-selection operations and one division by 2.

Computation of the extrinsic information:

The computation of the extrinsic information for each symbol δ is then performed with 2^m additions and 2^m subtractions according to (1-66). It is assumed that the addition of the *a priori* information with the metric $\sum_{l=1}^{m} v_{k,l} \cdot u_{k,l}$ corresponding to the information symbols has already been computed in the branch metric computation step (1-58).

Overall complexity:

Table 1-2 summarizes the resulting complexity corresponding to a trellis stage (or equivalently to an information symbol) of the Max-Log-MAP algorithm. Adding the respective complexities of the computations of the branch metrics, the forward and backward recursions, and soft output computations leads to a complexity of $3.2^m(2^\nu+1)+ 8(n-1)+2^{m+1}$ additions and $2^\nu(3.2^m-2)-1$ comparison-selection operations. For the RSC code (3,2,3) of Figure 1-7, the computational complexity amounts to 131 additions and 79 comparison-selection operations.

	Addition	Comparison-selection	Multiplication
Branch metrics (forward or backward)	$n_{ADD\_BRANCH}(n)+2^m$, where : $n_{ADD\_BRANCH}(n) = 4 \cdot C_n^2$ for n = 2, 3, 4		2^m
One step recursion (forward or backward)	$2^{\nu+m}$	$2^\nu(2^m-1)$	
Lambda computations	2^m2^ν	$2^m(2^\nu-1)$	
Hard decision		2^m-1	
A posteriori likelihood	$2^m(2^\nu+1)-1$	$2^m2^\nu-1$	2^m
Extrinsic information	2^{m+1}		2^m
Soft outputs	$2^m(2^\nu+3)-1$	$2^m2^\nu-1$	2^{m+1}
Total computation per information symbol	$3.2^m(2^\nu+1)+ 4(n-1)+2^{m+1}-1+ n_{ADD\_BRANCH}(n)$	$2^\nu(3.2^m-2)-1$	2^{m+1}

Table 1-2: Computational complexity of the Max-Log-MAP algorithm

Memory requirements:

This algorithm also requires storing N sets of state metrics (either forward or backward), which can represent an unaffordable amount of memory for large N. The next section presents scheduling techniques of the respective forward and backward recursions through the trellis that aims at reducing these memory requirements.

1.5.6 Scheduling of the BCJR algorithm

1.5.6.1 Overview

The equations of the Max-Log-MAP algorithm given in the previous sections can be implemented according to several schedules that differ in the sequencing of the basic operations: forward recursion, backward recursion and computation of the soft output. These schedules can also be applied to the original MAP algorithm, the Log-MAP algorithm or the Max-Log-MAP algorithm. For convenience, we will refer to the equations of the Max-Log-MAP algorithm.

1.5.6.2 Description of straightforward schedules

From the expression of the forward-backward algorithm, several schedules can be devised. In the first step, three particular cases are considered for convolutional RSC codes terminated with tail bits. These schedules are presented in Figure 1-15, where we adopt the same graphical representation proposed in [Boutillon *et al*.-03]. The horizontal axis represents the time τ, with units of a symbol period. The symbol period corresponds to the time needed to compute one stage of forward or backward recursion. The vertical axis represents the indices of the stages of the trellis. It is assumed that the soft inputs (observation data and *a priori* information) related to the N input symbols are stored in a memory and are all available from time $\tau = 0$.

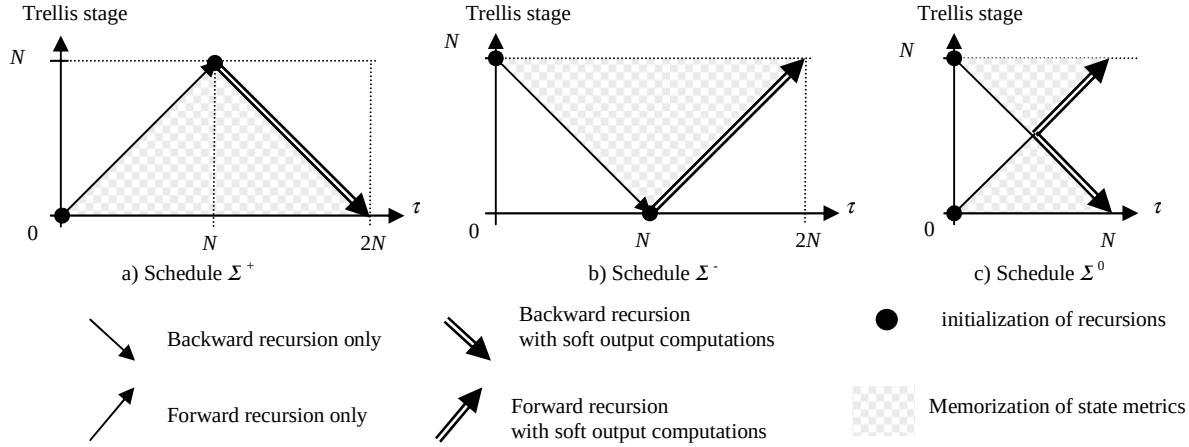


Figure 1-15: Schedules of the forward-backward algorithm for a trellis terminated with tail-bits: a) schedule Σ^+; b) schedule Σ^-; c) schedule Σ^0

Schedule Σ^+:

The first schedule (Figure 1-15-a) first computes the forward recursion according to (1-60) and stores all the forward state metrics that are generated. Then, once the forward recursion is completed, the backward recursion is computed using (1-61). Along with the backward recursion, the soft outputs are produced by combining the current backward state metrics and the forward state metrics following (1-62) and (1-66). Since the trellis is terminated with tail-bits, the starting and ending state correspond to the "all zero" state. The recursions are initialized accordingly.

This first schedule is denoted Σ^+ since the backward recursion is processed after the forward recursion. The decoding delay defined as the time needed to produce N soft outputs equals $2N$, and the size of the memory required for storing the forward state metrics amounts to N sets of states metrics.

Schedule Σ^-:

The second schedule (Figure 1-15-b) denoted Σ^- is the inverse of schedule Σ^+: the backward recursion is computed first, and then the soft outputs are produced along with the forward recursion. This schedule has the same decoding delay and memory requirements as

Σ^+. Another difference lies in the fact that the soft inputs are read in decreasing order from symbol N-1 down to symbol 0, and the soft outputs are produced in increasing order from symbol 0 to N-1.

Schedule Σ^0:

In order to reduce the decoding delay, another alternative can be used (Figure 1-15-c). In this schedule, the forward and backward recursions are computed concurrently. This explains the denomination Σ^0. From time $\tau = 0$ to time $\tau = N/2$, the forward and backward recursions process symbols 0 to $N/2 - 1$ and N-*1* down to $N/2$, respectively, and store the corresponding forward and backward state metrics. The two recursions cut across each other at time $\tau = N/2$. Then, from time $\tau = N/2$ to N, the soft outputs are produced along with the two recursions: the forward (backward) state metrics produced by the forward recursion (respectively backward) are combined to the previously stored backward (respectively forward) state metrics.

Note that this third schedule Σ^0 has the same computational complexity and memory requirements as the two others, Σ^+ and Σ^-. The decoding delay is halved to N cycles, however. This reduced decoding delay comes at the expense of a more complex implementation, since two soft outputs for two symbols are computed concurrently. This aspect of the association between hardware implementation and scheduling will be addressed in detail in the next chapter.

Initialization of recursions for CRSC codes:

The above mentioned-schedules have been described for terminated RSC codes using tail bits to close the trellis to the "all zero" state. Hence, the forward and backward recursions are initialized to a 0 starting state. To decode tail-biting codes, the associated trellis is circular, *i.e.* the encoder ends in the same state as the initial state, called the circulation state S^c, which is not known by the decoder. Since the optimal MAP algorithm provides the soft outputs by considering all possible paths in the trellis from the initial to the final state, a straightforward method of accomplishing this decoding is to perform a different forward-backward algorithm following (1-62) once for each of the 2^ν states. Then, a marginalization of the partial APPs over all the starting states leads to the final APP. This represents a dramatic increase in the decoder complexity, becoming 2^ν times the complexity of the algorithm given in the previous section.

Fortunately, there exists a sub-optimal method described in [Berrou *et al*.-99a], which evaluates the circulation state thanks to a pre-processing step. This pre-processing step uses the property of convergence of the concurrent paths using the Viterbi algorithm. The recursion is initialized to a uniform probability distribution, *i.e.* all states are initialized to the same value. After L' steps of a Viterbi recursion along the trellis, the end states of all paths converge to the same state. In conjunction with the circular property of the trellis, this method is used to evaluate the circulation state as follows (see Figure 1-16). A pre-processing step is performed in the backward direction using symbols with indices L'-1 down to 0. The obtained final state metrics are used to initialize the backward recursion. Likewise, a pre-processing step is performed in the forward direction using symbols N-L' to N-1, and the final state

metrics initializes the forward recursion. This sub-optimal algorithm has been described with schedule Σ^-, but it can also be extended to the other two schedules.

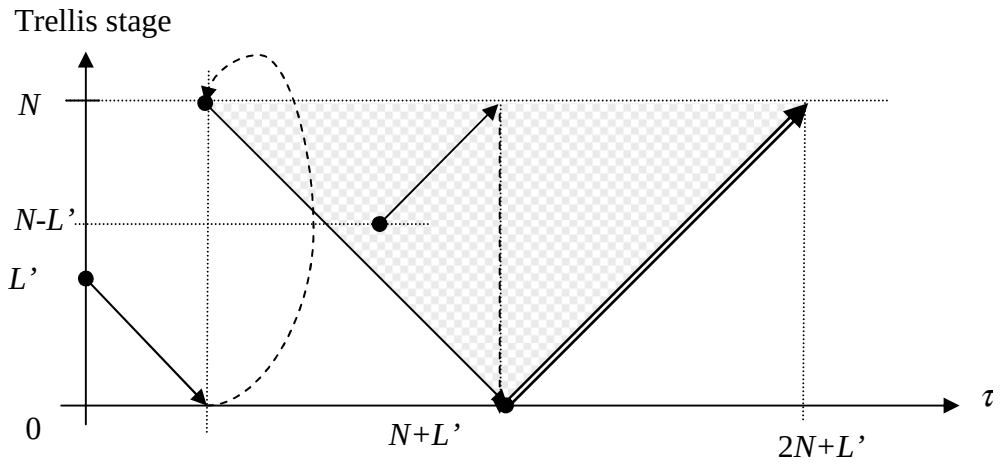


Figure 1-16: Initialization of the recursions using a pre-processing step of *L'* stages with schedule Σ^-

Since schedules Σ^+ and Σ^- are similar in terms of performance, decoding delay and complexity, only schedule Σ^- is considered in the remainder of this manuscript. Additionally, the schedules are described in the case of CRSC codes.

1.5.6.3 Sliding window scheduling

Both schedules Σ^0 and Σ^- suffer from two drawbacks: the decoding delay and memory requirement grow linearly with the length of the frame *N*. For large values of *N* the decoding delay and required memory might become critical or even unaffordable. To overcome these limitations, sliding window algorithms were proposed as an approximate version of the forward backward algorithm in [Benedetto et al.-96b] and then in [Viterbi-98]. The decoding length is then limited to a given truncated length *L* rather than the frame length *N*. In general, the value of *L* is independent of *N* and much lower than *N*, yielding a significant improvement in memory utilization and decoding delay. The choice of *L* trades off the complexity reduction and the performance degradation due to the sub-optimality of the backward recursion.

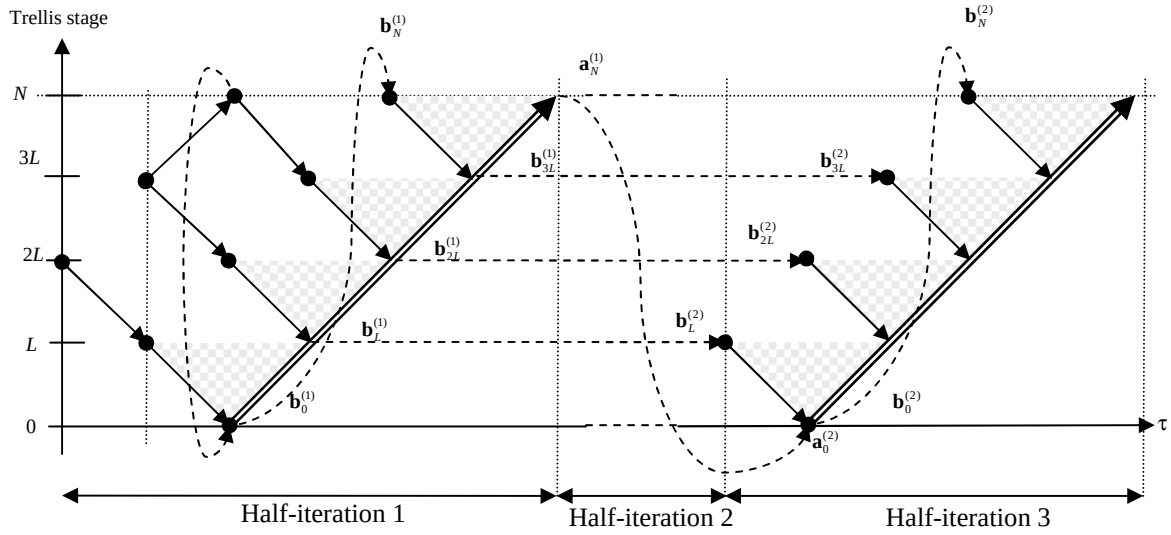


Figure 1-17: Sliding window schedule *SW-Σ⁻* with a pre-processing step to initialize the recursions at the first iteration

Basically, the decoding process of a sliding window algorithm using schedule Σ^- for a tail-biting code can be described as follows (see Figure 1-17). The forward recursion is performed continuously using symbols with indices 0 to N-1. A sliding window algorithm is applied to the backward recursion, which is truncated into several independent and consecutive recursions of size L. This truncation exhibits windows of size L on which a schedule Σ^- is performed. This new schedule is thus denoted $SW\text{-}\Sigma^-$.

Initialization of recursions:

The initialization of the recursions is illustrated in Figure 1-17, in the case of a turbo decoder built with two RSC encoders. For the sake of clarity, the processing of only the first and third half-iterations (corresponding to decoding in the natural order) is represented. During the second half-iteration, the decoding of the interleaved order is performed with the same schedule as the first half-iteration.

The recursions are initialized using a pre-processing step. For the forward recursion, only the first window requires a pre-processing step on the L' last stages of the trellis. For the other windows, the forward recursions are initialized by the final forward state metrics of the previous window. For the backward recursions of all windows, except the last one, the recursions are initialized by a pre-processing step of size L'. For the last window, it is initialized with the final backward state metrics of the first window computed previously.

In terms of complexity, this initialization scheme using pre-processing steps results in an increase corresponding to the computation of $N' = (N/L) \cdot L'$ recursion stages for each half-iteration. Taking for convenience $L' = L$, the additional complexity is $N \cdot 2^{\nu+m}$ additions and $N \cdot 2^{\nu}(2^m-1)$ comparison-selection operations. In the context of iterative decoding, an initialization scheme of lower computational complexity has been proposed independently by Dingninou *et al.* in [Dingninou *et al.*-99] and [Dingninou-01] and Dielissen *et al.* in [Dielissen *et al.*-00]. For this scheme, the pre-processing steps are only performed for the first iteration. Then for the subsequent iterations, the recursions are initialized by the

corresponding final state metrics of the previous iteration. For a circular trellis, the forward state metrics $\mathbf{a}_0^{(i)}$ are initialized with the final forward state metrics $\mathbf{a}_N^{(i-1)}$ obtained at the previous iteration. Likewise, the initial backward state metrics $\mathbf{b}_{w \cdot L}^{(i)}$ of window w are initialized with the final state metrics $\mathbf{b}_{w \cdot L}^{(i)}$ of the next window. The memory requirement for storing these state metrics is $2\lceil N/L \rceil \cdot 2^{\nu}$ bits for a 2-dimensional turbo code.

This latter scheme called Next Iteration Initialization (NII) introduces a trade-off between computational complexity and memory storage. Indeed, for two-dimensional turbo codes, $2 \cdot \lceil N/L \rceil \cdot 2^{\nu}$ state metrics are stored, in addition to the $L.2^{\nu}$ state metrics for the forward-backward algorithm of each window. A comprehensive survey on the choice of L has been conducted in [Dingninou-01]. To summarize, the choice of the sliding window length L influences the dimensioning of: the memory size for the state metrics, the memory size for the initialization of the state metrics, the computational complexity, the decoding delay and the performance of the decoder. The selection of a value for L (generally fixed for a given implementation) has to lead to the best possible trade-off.

Further complexity reduction:

A further computational complexity reduction can be achieved by suppressing the pre-processing steps of the first iteration. In this case, the reliability of the extrinsic information produced by the first iteration is decreased. In order to compensate for this lower reliability a pragmatic and simple approach involves scaling the associated extrinsic information using a low scaling factor, such as 0.5 for instance (see section 1.5.4 for details about extrinsic information scaling).

At equivalent computational complexity, the performance of this variant is equivalent or even better than the scheme with pre-processing at the first iteration. Actually, the computational complexity associated with the pre-processing steps equals the complexity of one recursion over N trellis stages. Hence, the pre-processing steps of the first half-iterations (one half-iteration for each dimension) amounts to two complete recursions. With the enhanced scheme, one more half-iteration can be performed, which explains the improvement in performance.

A sliding window principle can be applied to the other forward-backward schedule Σ^0 leading to the schedule *SW-*Σ^0. The two schedules *SW-*Σ^- and *SW-*Σ^0 are illustrated in Figure 1-18 for a single half-iteration.

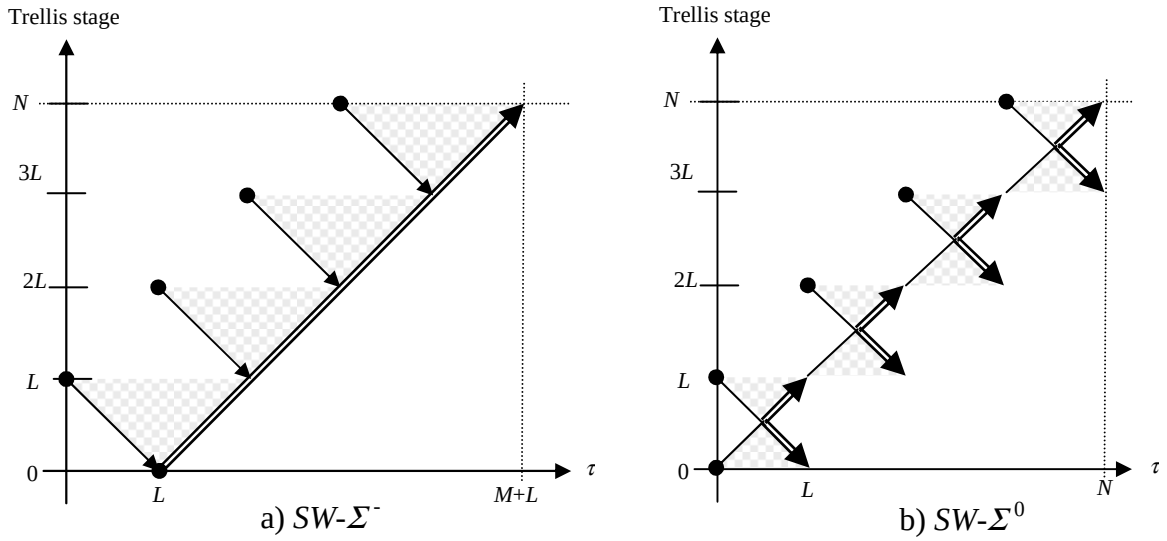


Figure 1-18: Sliding window schedules for one half-iteration: a) *SW-*Σ^-; b) *SW-*Σ^0

1.6 Conclusion

In this first chapter, we have recalled the basics for error correcting codes and especially convolutional codes and turbo codes. Our description has concentrated on the parallel concatenation of rate *m/n* Circular (tail-biting) Recursive Systematic Convolutional codes. One important component underlying the performance of turbo codes is the interleaver. We have described several key interleaver designs among the very abundant literature available. They share a common objective: to improve the bit error rate of the associated code in the convergence region (at low SNRs) and/or in the error floor region (at high SNRs).

We have also summarized the state-of-the-art in decoding algorithms for turbo codes. After a description of the iterative decoding algorithm, the role of soft-input soft-output algorithms associated with extrinsic information has been discussed. A survey on soft output decoding algorithm for double-binary RSC has been carried out. The comparison has led to the supremacy of the Max-Log-MAP algorithm yielding the best trade-off complexity against performance. The complete derivation of the Max-Log-MAP algorithm for double-binary RSC codes was then presented together with its computational complexity.

Finally, schedules for a practical implementation of the Max-Log-MAP algorithm have been described and compared. Schedules using a sliding window approach are of great interest, since the decoding delay and memory requirements are reduced. The conjunction of the Next Iteration Initialization technique decreases the computational cost to the minimum possible: two Viterbi recursion stages and one soft output computation stage per symbol per half-iteration.

Now that we have seen the potential of turbo codes (performance within 1 dB of the Shannon capacity) and their iterative decoding algorithm, the question arises as to how to implement efficiently a high-throughput turbo decoding architecture. This concern will be addressed in the next chapter. In particular, we will show that the two problems of finding powerful turbo codes and efficient decoding algorithms are closely linked together. More

precisely, it will answer two observations raised in this first chapter. The first one concerns the design of the interleaver for the performance of the turbo code. We will demonstrate that the design of the interleaver from a solely performance view leads to the main architectural constraints on the implementation of a turbo decoder. Second, we have observed that for the same computational complexity and memory requirements, schedule Σ^- has a greater decoding delay than schedule Σ^0. This will be explained by the difference in hardware complexity.

Chapter 2

Hardware architectures for high throughput turbo decoding

Contents:

Chapter 2. Hardware architectures for high throughput turbo decoding

In the previous chapter, we described the principles of turbo codes and turbo decoding. This study led us to the consideration of the simplest expression of the Max-Log-MAP algorithm, whose complexity has been evaluated. This second chapter introduces the context of the implementation of high-throughput turbo decoders. In particular, we will describe a general framework of architecture optimization that will be applied to turbo decoder architectures in order to identify the major research directions we have followed in this thesis.

In the first section, the concept of architecture is introduced along with the motivations and the solutions for the optimization of its efficiency.

There follows in the second section a description of state-of-the-art architectures for turbo decoding using the Max-Log-MAP algorithm described in the previous chapter. The application of the architectural optimization techniques to turbo decoder architectures is then addressed, through the use of a simplified model of "architecture efficiency" suited to a variety of turbo decoders architectures, and especially oriented to high-speed concerns. This model constitutes an original contribution of our work. In particular it links the relation between the schedule of the SISO algorithm and the resulting complexity of a given architecture by introducing the concept of activity.

This model identifies the different contributions to the architecture throughput and efficiency offered by critical design parameters and architectural choices. It can thus help at quantizing the optimization space left to improve the throughput of turbo decoders and the efficiency of their architectures. These approaches are discussed in the third section.

2.1 Fundamentals of hardware architectures

In this section, we will first introduce the general concept of architecture. Then, we will present the hardware circuits on which these architectures are generally implemented. After explaining the interest of optimizing an architecture, we will derive useful measures used to evaluate an architecture. Finally, a general model of architecture efficiency is introduced and will be used to derive means for improving its throughput and efficiency. These means will be illustrated thanks to a simple example.

2.1.1 What is an architecture ?

An architecture involves the description of the manner in which a system (infrastructure, hardware and software) is realized. An architecture usually describes how the system is constructed, what are the resources that are used, and how these resources interact with each other. A system has architectures at different levels of description, corresponding to different

implementation layers as described in Figure 2-1. Each functional description level of the system corresponds to an implementation layer, organized in a hierarchical way. At the lowest level of the functional description, functions are described at the bit level (0 or 1) and the corresponding implementation is described with transistors. The aggregation of several bit level operations increases the level of description which results in boolean functions (AND, OR, NOT, etc.) implemented with gates. Then, grouping boolean functions into elementary operations (additions, multiplications, comparisons, etc.) corresponds to functional blocks (adders, multipliers, comparators, etc.). Increasing further the level or description leads to more complex operations, process and complete systems corresponding at the implementation layer to integrated circuits, boards, and complete systems, respectively. Note however, that in the context of System On Chip (SOC) more and more implementation layers are implemented on a single chip. SOC are already implementing all layers up to the complete system. And, the implementation of complete networks on a single chip is also studied in the literature.

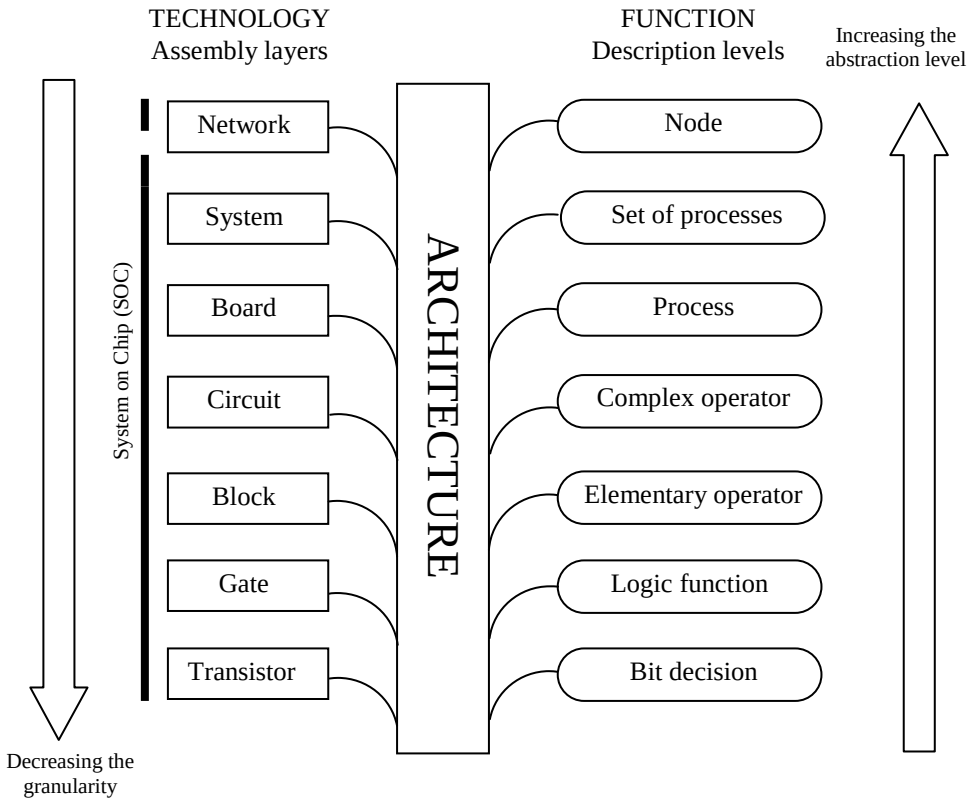


Figure 2-1: Description levels and corresponding implementation layers of a system

The term "architecture" is used here to refer to the description of a set of hardware resources (computation components, memorization components, communication components, etc.) that will be implemented in a circuit. This description also specifies the execution of tasks of the algorithm on this architecture and the associated execution schedule of tasks. It is based on a double allocation of resources:

- a spatial allocation that assigns the execution of each task among several parallel tasks to a resource of the architecture.
- a temporal allocation that specifies the succession of tasks assigned to each resource.

This allocation in time and space of resources can be performed at each level of description. Optimizing an architecture involves instantiating the minimal set of resources to which the tasks of the algorithm can be allocated, according to the possibilities and constraints of each description level. To do that, one has to maximize its efficiency, which is defined as the ratio between the number of resources strictly required and the resources effectively instantiated.

Before establishing metrics for evaluating an architecture, the following section gives the circuits on which it is implemented.

2.1.2 What is the target of an architecture ?

Implementation of an architecture can be done on several different circuits, with specific characteristics, performance and constraints. Generally, an algorithm for digital signal processing is implemented on a Very Large Scale Integration (VLSI) circuit. VLSI refers to the integration of transistors on a substrate of silicon that are connected through wires and performing a given function. This integrated circuit is called a chip. Integration of transistors on a single chip started after the invention of the transistor in 1948, with the invention of the planar process in 1959 for producing transistors on a single substrate. This evolution enabled more and more transistors to be integrated implementing more and more complex operations. Nowadays, a single chip contains several hundreds of thousands transistors per mm^2.

Several different classes of circuits can be implemented on a VLSI circuit, but they are all composed of three key elements:

- data operators: gates that implement the operations required by the algorithm.
- storage elements: registers, and memory blocks that save and store the data of the algorithm. A memory block can be for instance a Random Access Memory (RAM) or a Read Only Memory (ROM).
- wires: point-to-point wires, buses, that communicate values between and among the storage elements and data operators.

These key elements are provided in various conceptual and/or physical packages. In our context, only three different concepts are considered. The design of the architecture is based on either

- a software implementation on programmable general purpose processor such as Digital Signal Processors (DSP), or
- a hardware implementation on an Application Specific Integrated Circuit (ASIC), or
- a hardware implementation on a Field Programmable Gate Array (FPGA).

The choice of one circuit among the other depends on several parameters: cost, speed, power consumption, complexity, flexibility, development time, etc. The relations between the classes of circuits and these parameters are then described.

Digital Signal Processors:

A DSP has fixed resources (data operators, storage elements and wires) that are programmed by the designer in order to implement the algorithm. The programmability functionality is brought by the presence of a processor. The basic modules of a DSP are represented in Figure 2-2.

A processor comprises a control unit whose evolution is directed by a list of instructions called a program and stored in a memory. Another memory also stores the data required for the execution of the algorithm. In order to execute computations, a processor comprises an Arithmetic and Logic Unit (ALU), which enables operations such as additions, comparisons, logical operations, etc to be performed. DSPs are processors with additional computational units dedicated to signal processing (see Figure 2-2). Their hardware architecture is characterized by the size of the memories, the number of processing units, etc. For programmable devices, the instructions of a program are executed sequentially, and the task of the designer is then to write the program that will be compiled and executed on the processor. For devices with multi-processing units, the task of the designer (and the compiler) is more complicated. The execution of an algorithm on a multi-processing unit processor requires allocating the resources to the computational units. A more efficient allocation of the operation can be achieved by the knowledge of the algorithm. Therefore, in the context of a programmable processor, the design of an architecture specifies the organisation of the program: memory organization, communications between the memory and the processing units, utilization of resources, etc.

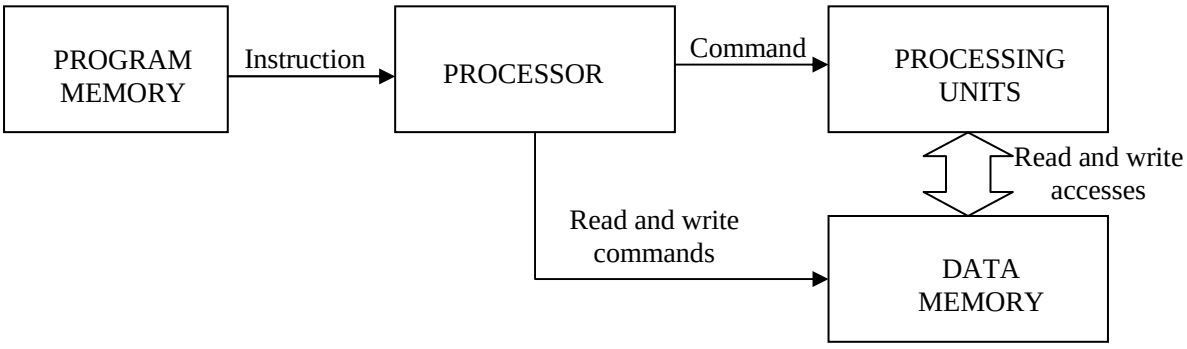


Figure 2-2: Description of a Digital Signal Processor

However, due to their sequential processing, DSPs are not well suited for applications demanding high throughput. On the contrary, hardware implementations are more suited in this case since the parallel operations of the algorithm can be processed simultaneously by dedicated resources.

ASIC:

An ASIC integrates the resources required by the execution of the application on a single chip. They are selected by the designer according to the requirements of the application, and are thus specific to this given application (explaining their name "Application-Specific" IC).

The chip technology family we consider here is called "standard cell", which uses a cell library containing basic pre-designed logic gates and elementary operations. The resources

required by the algorithm are implemented with these standard cells. On top of the silicon of which the cells are made up with, the wires are materialized with several layers of metallization connects the cells together.

An ASIC can also integrate on-chip synchronous memory blocks such as RAMs or ROMs. A memory block, also called a memory bank, is defined as a table of values accessed by specifying its address through an access port. The memory content at the specified address is provided on the data bus of the port for a read access. For a write access, the value presented on the data bus is written into the memory at the corresponding address position. Concurrent read and write accesses to different addresses cannot be performed simultaneously through a single-port memory. To this end, some memories can have additional ports, generally limited to 2, in order to allow independent accesses. The silicon area needed for a double-port memory bank is around twice the area of a single-port bank. Single and double-port memory banks are represented in Figure 2-3. Each port contains its own data and control signals: clock (CK), read enable (RDEN), write enable (WREN) and address (ADDRESS).

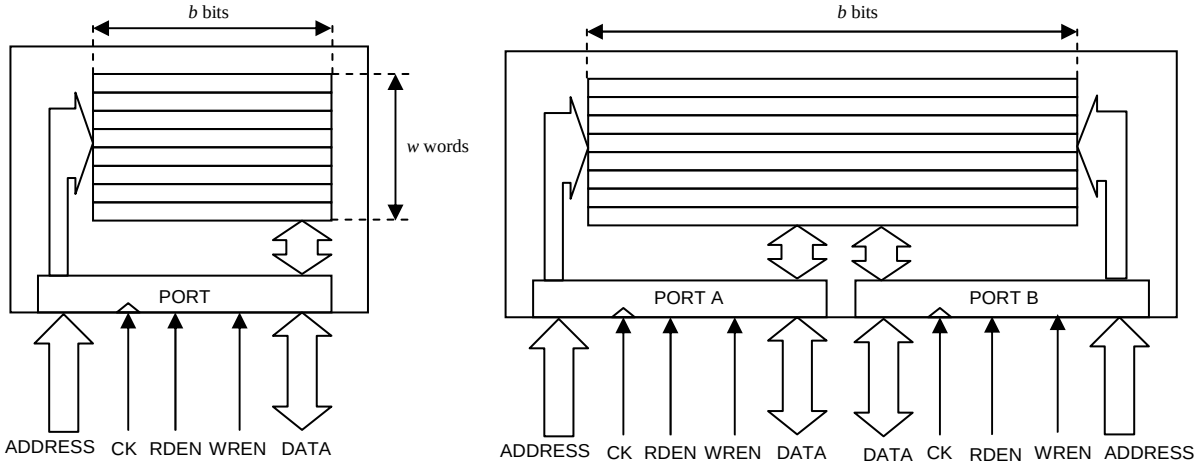


Figure 2-3 : Single-port and dual-port memory banks for memory of depth *w* words and width *b* bits

A memory bank contains *w* words, each of width *b* bits. The number of bits $b \cdot w$ of a memory block is the primary parameter that influences its area. Nevertheless, the ratio between the depth (number of words) and width (number of bits per word) of the memory block has also a non-negligible impact on the area. This ratio is referred to as the form factor of the memory block. Let us consider two memory blocks with the same number of bits $a \cdot b$, $a >> b$. The memory synthesis experience tells that the memory with depth *a* and width *b* has a smaller area than the memory with depth *b* and width *a*. As a general rule, it is better to use deep memory blocks with a reduced width.

Production of ASICs is costly both in time and money. Nevertheless, it represents the best solution in terms of speed, area and power consumption. The other disadvantage of an ASIC implementation is that it is not flexible, since there is no programmability capability. An ASIC is for these reasons generally designed for applications requiring large quantities and/or high computational throughput.

FPGA:

Field Programmable Gate Arrays belong to the family of programmable hardware circuits. In fact, an FGPA integrates the three key elements on the chip: logic, storage and wires in a standard part that can be programmed.

An FPGA is a very regular structure constructed around a matrix of functional units as shown in Figure 2-4. The functional units and wires are configured by a configuration layer, which is generally made of a Static RAM (SRAM). This layer sets up gates, storage elements and interconnect paths: it brings the programmability capability to the FPGA. FPGA families are differentiated by their chip-level architecture, the granularity of their functional unit and their wiring organization. The two families that are considered in this manuscript are the FPGAs provided by two leading companies in the FPGA market, namely Xilinx and Altera. For both families, the functional part is made of a LUT with multiple inputs, generally 4 bits[1] and a single output bit. It enables any function of four variables to be generated. Each LUT is also associated with a programmable register and a carry chain with carry select capability. This association constitutes the smallest unit of logic in an FPGA and is called a Logic Element (LE), represented in Figure 2-4. The carry chain is used to implement efficiently operations such as additions or subtractions where the carry resulting from the addition of two bits is propagated from the Least Significant Bits (LSB) to the Most Significant Bits (MSB).

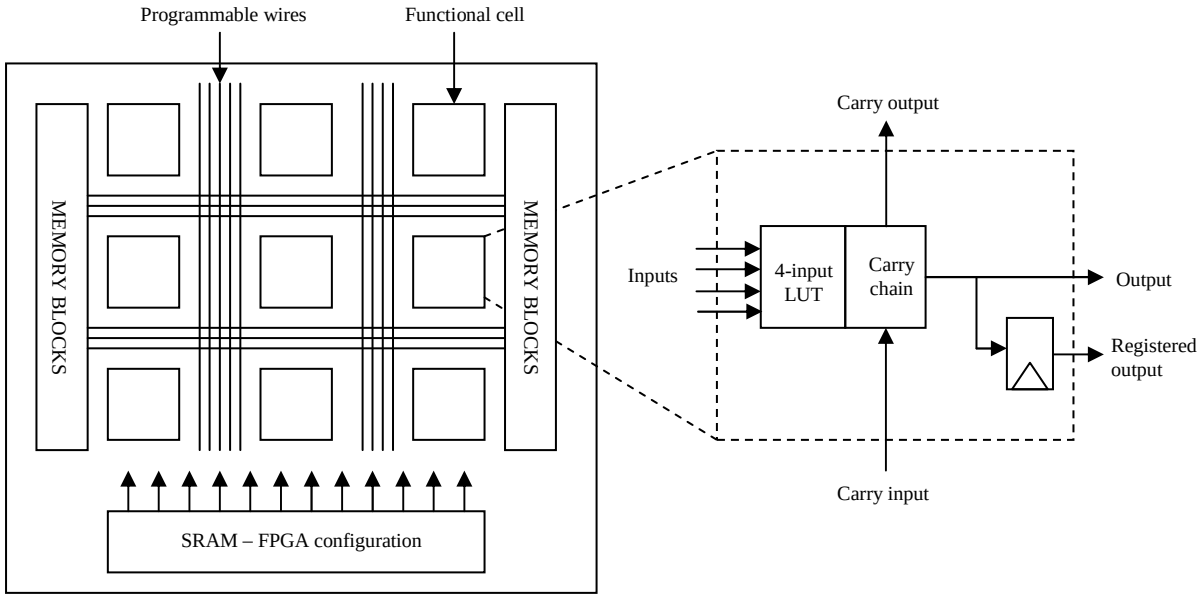


Figure 2-4: Conceptual structure of FPGAs

Generally, the memory blocks of the FPGA have two ports. A single memory block can be programmed in several different configurations of width and depth. The form factor also plays a crucial role in the number of memory blocks required to implement a logic memory. A logic memory is referred to as the memory required by the algorithm, which does not specify on which memory blocks it is mapped in the FPGA. For example, let us consider an FPGA with 4 Kbit memory blocks of maximal width 32 bits, with an associated depth of 128 addresses. If

[1] Recently, this choice has been altered. For instance, the lastest family of Altera, Stratix II includes a functional part with 8 inputs, that can be used in a very scalable way.

the algorithm requires a logic memory of 32 addresses of 128 bits (*i.e* 4 Kbits), then the number of required memory blocks equals 4 (and not 1, as one would expect).

In our work only ASIC and FPGA circuits have been considered. However, some techniques developed for these circuits have been applied successfully to the design of a turbo decoder on a DSP with multiple computation units. The results of this study are presented in [Gnaedig *et al*.-04], which is reproduced in Appendix E.

2.1.3 Why optimize an architecture?

The main objective in optimizing an architecture is to decrease the costs associated with its implementation (hardware cost) and the costs associated with the execution of tasks on this architecture (running cost).

At the circuit level, the hardware costs are generally related to the complexity of the architecture (area, number of computational resources,etc.). These costs greatly depend on the implementation circuits of an architecture. For instance for an ASIC, the area of the chip is the primary factor that drives its cost. The cost of circuits varies indeed exponentially with its area [Ullman]. For a given FPGA, the maximum number of LEs bounds the maximum complexity of a system that can fit into it. This bounding of the available resources explains the importance of the optimization step. If the complexity of an architecture slightly exceeds the available resources of a given FPGA, then a larger FPGA has to be selected, which can represent a significant cost increase. Alternately, there might be examples were a small complexity reduction makes it possible to fit the architecture into a smaller FPGA.

Regarding the running costs of an architecture, they can be associated for instance with power consumption. For mobile applications, the power consumption of an architecture has a great impact on the costs of the whole system, since high power consumption requires a large battery.

In order to optimize an architecture, realistic cost functions should be derived to evaluate fairly different architectural solutions. Cost functions also make it possible to draw fair comparisons between several different architectures.

2.1.4 How to evaluate an architecture

Evaluation of an architecture can take different forms, depending on one’s interests. For instance, someone with a practical objective would be primarily interested in observable performance. This could include multiple dimensions of evaluation, involving speed, latency, area, power consumption, flexibility, adaptability, accuracy, functionality, etc. The architecture is then optimized regarding one or more of these performance measures.

In our context of hardware implementation of turbo decoders, we are primarily interested at their error decoding performance. The comparison of architectures for turbo decoding is made assuming an equivalent error decoding performance. In this case, three additional

evaluation criteria are considered: complexity and throughput in the first step, latency in the second step. Power consumption will not be taken into account in this comparison[1].

The yardstick of complexity:

The complexity of the architecture can be measured either by the area (in mm²) for a silicon integration, the number of transistors, the number of gates, the number of LEs for an FPGA, the number of elementary operators (adders, comparators, etc.) or more complex operators resulting from the aggregation of several elementary operators. The choice of granularity in the measure of complexity depends on the implementation circuit of the architecture and the implementation level considered. This granularity concept in the complexity measure assumes that all resources (transistors, LEs, elementary operators or complex operators) have equivalent complexities. This assumption has to be verified by synthesis of the resources into a description at a lower implementation layer. For example, with an ASIC implementation, it is assumed that the complexity of an adder is equivalent to the complexity of a comparator. This can be verified by comparing the area in mm² of the two components after logical synthesis.

Obviously, these different hardware complexity measures are not equivalent. For example, the complexity expressed in number of transistors cannot give an exact area of the corresponding chip in mm², since it does not take into account the complexity of the memories and of the communication wires. Consequently, the level at which the comparison is carried out is chosen according to the difference in the resource usage between the architectures considered. In particular, resources that are common to the evaluated architectures can be extracted from the comparison. For instance, under the assumption that two algorithms are equivalent in terms of memorization and communications, their complexities can also be compared by counting the number of transistors. At a higher level, two architectures that have the same complexities in terms of registers can be compared by an counting their elementary operators.

Throughput:

The throughput of an architecture expressed in number of samples processed per second is defined as

$$D_s = \frac{n_S}{t_s} \text{ [samples/s]}, \qquad \textbf{(2-1)}$$

where n_S is the number of samples produced by the architecture during t_S seconds. It is expressed in number of samples per second. A sample is defined as an output of the architecture, which can be for instance a bit, a word, a vector, etc.

Latency:

The latency is commonly defined as the maximum delay between the input of a sample and the production of the corresponding output.

[1] Reducing the area of the chip reduces the power consumption since area and power consumption are proportional. But, power consumption also depends on other parameters such as activity of the operators, frequency of accesses to the memories and communications.

Efficiency of an architecture:

Assuming equivalent decoding performance, increasing the throughput of an architecture is the major objective of the design process. However, increasing the throughput of an architecture regardless of its complexity is not efficient. An efficient architecture aims to maximize the throughput with minimal complexity. Hence, the ratio between the throughput and the complexity is chosen as a measure for the evaluating its efficiency. The optimization process leads to the maximization of this ratio. The means to achieve this goal are presented in the next section.

2.1.5 Means for improving the efficiency of an architecture

Improving the efficiency of an architecture can be achieved by several means. Clearly, one can either increase the throughput at constant complexity or decrease the complexity at equivalent throughput.

2.1.5.1 Hardware complexity reduction

Let us first discuss the reduction in hardware complexity. The area of the architecture corresponds to the cumulative area of the computational operators, the area of the memories and the area of the interconnections. Several degrees of freedom are available to the designer to optimize the complexity using the three following architectural laws:

- Law 1: Computation and memorization can be exchanged against each other.
- Law 2: Computation and interconnections can be exchanged against each other.
- Law 3: Memorization and interconnections can be exchanged against each other.

The existence of such transformations depends on the algorithm considered. The interest of using such transformations to reduce the complexity depends on the implementation circuit of the architecture on the characteristics of the algorithm. For a VLSI circuit, the memory generally represents a large proportion of the circuit, since the implementation of a memory of 1 bit requires several transistors. Hence, for an algorithm with high memory requirements, the designer aims to decrease the latter by using more computations. On the other hand, with an FPGA implementation, the memories are already implemented on chip. For some applications, the available memory is in excess and the computation requirements are the limiting factor for fitting the architecture into the FPGA. In this case, transformations in the algorithm can decrease the computation requirements by using more memories.

2.1.5.2 Throughput augmentation

Then, second, the increase in throughput is studied, also taking into account the associated evolution in complexity.

Let us consider a synchronous architecture, controlled by a clock called *clk*. A simple definition of a synchronous architecture is one whose actions are governed by a clock. It is a signal that oscillates continuously. The architecture can only perform operations at discrete points in time, corresponding to the edges (often, only the clock’s rising edge is considered) of the clock's cycles. A synchronous architecture is composed of combinational gates and

registers. All the registers of the architecture are driven by a single clock *clk*, which controls the sampling of the input data: on a rising edge of the clock, the value at the input of the register is transferred to the output of the register, thus enabling memorization of the input for the duration of one clock cycle. The maximum operating frequency f_{clk} of an architecture is given by the longest propagation delay t_{clk} between two registers. The corresponding path between the two registers is called the critical path. Hence, for a synchronous architecture, the time t_S in (2-1) is expressed as a multiple of the clock period $t_{clk} = 1/ f_{clk}$, and the throughput is then given by

$$D_s = \frac{n_S}{n_c \cdot t_{clk}} = f_{clk} \frac{n_S}{n_c} \text{ [samples/s]}, \quad \textbf{(2-2)}$$

where n_c is the number of cycles required to produce n_S samples.

Let us consider that the architecture implements Γ computational resources of the same granularity. In other words, all the computational resources of the architecture are from the same implementation layer and have an equivalent complexity. Γ corresponds to the computational power of the architecture, *i.e.* the maximum number of operations that can be performed at each cycle. Let $E_r = n_c \cdot \Gamma$ be the maximum number of operations that can be performed during n_c cycles. The equation (2-2) can be rewritten as

$$D_s = f_{clk} \frac{n_S \cdot \Gamma}{E_r} \text{ [samples/s]}. \quad \textbf{(2-3)}$$

The different operations of the algorithm are allocated in space and time to these Γ resources. Let n_a denote the number of elementary operations actually required by the algorithm to compute one output sample. Hence, the processing of n_S samples requires $n_e = n_S \cdot n_a$ elementary operations. Introducing this last relation into (2-3) yields:

$$D_s = f_{clk} \frac{n_e}{E_r} \cdot \frac{\Gamma}{n_a} \text{ [samples/s]}. \quad \textbf{(2-4)}$$

Let $\alpha = n_e / E_r$ be the number of elementary operations required by the algorithm to produce n_S samples divided by the total computational resource available E_r during the production time allocated to these samples. This ratio is called the activity of the architecture and corresponds to the average utilization of the computational resources during the execution of the algorithm. Evidently, it depends on the algorithm and also on the resources that are implemented, and is thus denoted $\alpha(\Gamma)$. The derivation of the throughput is then as follows:

$$D_s = f_{clk} \cdot \alpha(\Gamma) \cdot \frac{\Gamma}{n_a} \text{ [samples/s]}. \quad \textbf{(2-5)}$$

From (2-5), three means for increasing the throughput of the architecture can be envisaged:

- Improvement in the utilization of the computational resources characterized by the activity α.
- Increase in the clock frequency f_{clk}.
- Multiplication of the number of computational resources Γ.

These techniques are illustrated in the following example.

Example 2-1:

Let us consider the computation of the function F depending on 8 operands v_1 to v_8, according to

$$F = F_2\left[F_1\left(F_1(v_1, v_2), F_1(v_3, v_4)\right), F_1\left(F_1(v_5, v_6), F_1(v_7, v_8)\right)\right] \quad \textbf{(2-6)}$$

where F_1 and F_2 are some sub-functions of two operands. This computation involves $n_a = 7$ elementary operations and can be represented by a data flow graph (DFG) depicted in Figure 2-5.

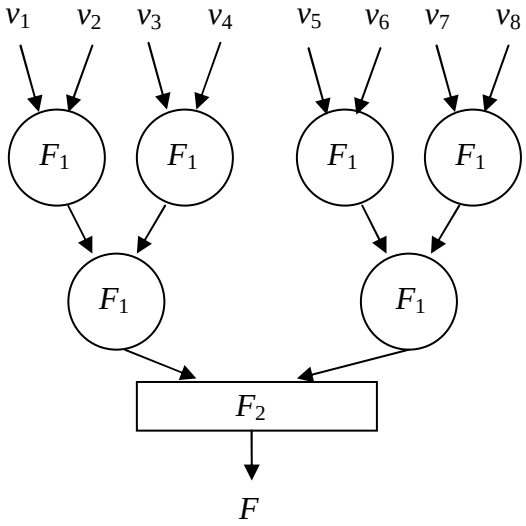


Figure 2-5: Data flow graph of the function *F*

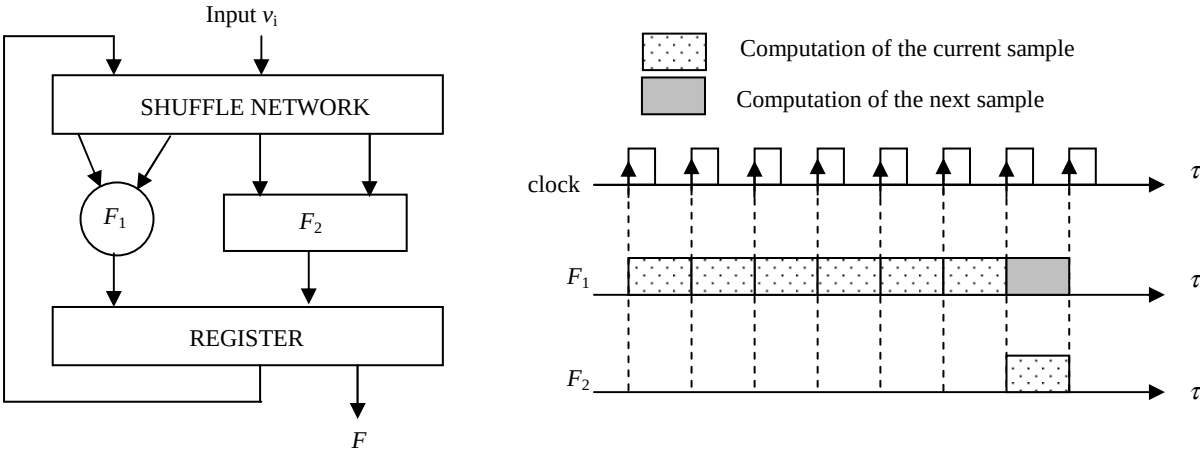


Figure 2-6: Allocation of resources for the computation of *F* with architecture ($\Gamma = 2$, $n_c = 14$, $n_s = 1$)

A minimal complexity architecture for the computation of F requires two operators F_1 and F_2 that are used successively. We consider in the first step that the two operators are of equivalent complexity. Then, the computational resource equal $\Gamma = 2$ (1 operator F_1 and 1 operator F_2). The allocation of the operations to the resources is given in Figure 2-6. Since the last operation involves only operator F_2, operator F_1 can be used to compute the next set of input data. A total of $n_c = 6$ cycles are thus needed to produce 1 output sample, resulting in a

computational energy of E_r = 12. A total of 7 operations are thus performed during the 6 cycles and the average activity of this architecture equals α = 7 / 12 = 0.58. The average throughput amounts to one sample every 6 cycles: $D_s = f_{clk}/6$.

Increasing activity:

In order to double the throughput, a straightforward technique would duplicate the architecture described in Example 2-1. This duplication is not efficient, though, because the activity is unchanged and rather low. As has been mentioned above, the activity depends on the number of resources Γ and on the algorithm, especially the data dependencies induced by the algorithm. Introducing local parallelism can have a very positive impact on the allocation of operations to resources by increasing their utilization, as illustrated in the following example.

Example 2-2:

Now, let us consider that with the same function as in Example 2-1, only the operator F_1 is duplicated, resulting in Γ = 3 (2 operators F_1 and 1 operator F_2). The resulting allocation of the operations to the operators is represented in Figure 2-7. With this architecture, the computation of F requires only n_c = 4 cycles with a computational power $E_r = n_c \cdot \Gamma = 4 \cdot 3 = 12$. The two operators F_1 are again used in the last cycle to process the next set of input data. The resulting average activity is increased to α = 7 / (3·3) = 0.78. The average throughput equals one sample every 3 cycles: $D_s = f_{clk}/3$. It is shown that without completely duplicating the architecture, a multiplication by 2 of the throughput has been obtained, thanks to a significant improvement in activity.

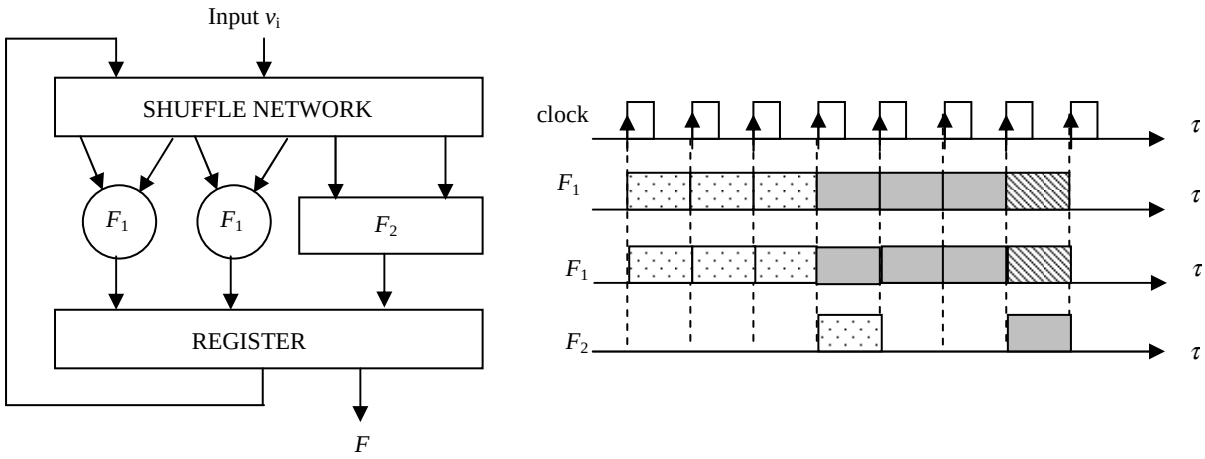


Figure 2-7: Allocation of resources for the computation of F with architecture (Γ = 3, n_c = 4, n_s = 1)

Increasing the clock frequency:

Increasing the clock frequency of an architecture is a technique commonly used, since it enables a proportional increase in the throughput. To do this, one can try to reduce the propagation time along the critical path. For VLSI design, this can be done by using faster operators. This increase in speed comes at a cost of considerably more logic, however. Another technique for reducing the propagation delay lies in using pipelining, *i.e.* inserting registers to cut the critical path into two or more parts. Pipelining increases latency, and also

complexity because of the additional registers. It is worth noting that pipelining is not always possible since, for some architectures, additional latency cannot be overcome. The next example illustrates the technique of pipelining.

Example 2-3:

Let us consider the full-parallel architecture for the computation of (2-6). A full-parallel architecture instantiates one dedicated operator for each operation of the algorithm. But, with this architecture, the critical path corresponds to the propagation through two operators F_1 and one operator F_2. Assuming identical propagation delays for F_1 and F_2, and assuming that the delay of a succession of operators is the sum of the elementary propagation delays, the clock frequency is therefore three times lower than the architecture of Example 2-2. Inserting two stages of pipeline registers as represented in Figure 2-8 enables us to triple the clock frequency. After a latency of l = 3 cycles, the architecture outputs one sample each cycle, if it is fed incessantly by new input data. The average activity tends towards 1 when n_S tends towards infinity. The average throughput equals $D_s = f_{clk}$, *i.e.* a multiplication by three compared to the architecture of the previous example.

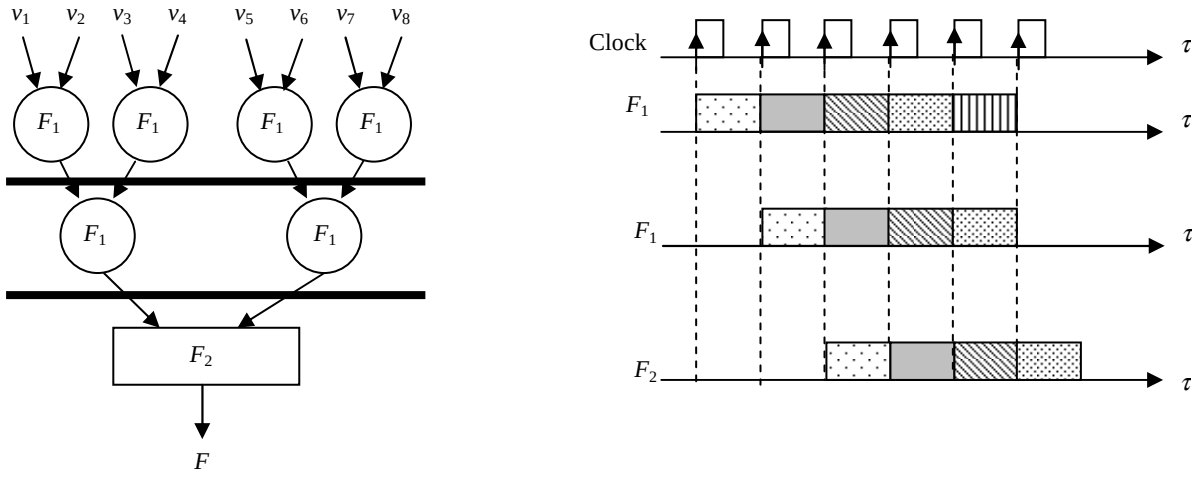


Figure 2-8: Full-parallel architecture (Γ = 7, n_c = 1, n_s = 1) for the computation of F

Multiplication of the number of resources:

Once the architecture is efficient (high activity) and the optimization techniques described above have been carried out, the eventual technique involves a global duplication of the computational resources. Increasing globally the parallelism degree of the architecture can be achieved by multiplying the number of resources by a factor ρ. Since this increase in parallelism comes at the expense of a multiplication of the hardware complexity by a factor of ρ, the efficiency of the architecture is obviously not improved: the ratio throughput divided by complexity remains constant. The next example applies this concept to the architecture of Example 2-3.

Example 2-4:

The architecture of Figure 2-8 is duplicated once in Figure 2-9. The average activity is unmodified at 1 and the throughput is multiplied by two: $D_s = 2 \cdot f_{clk}$.

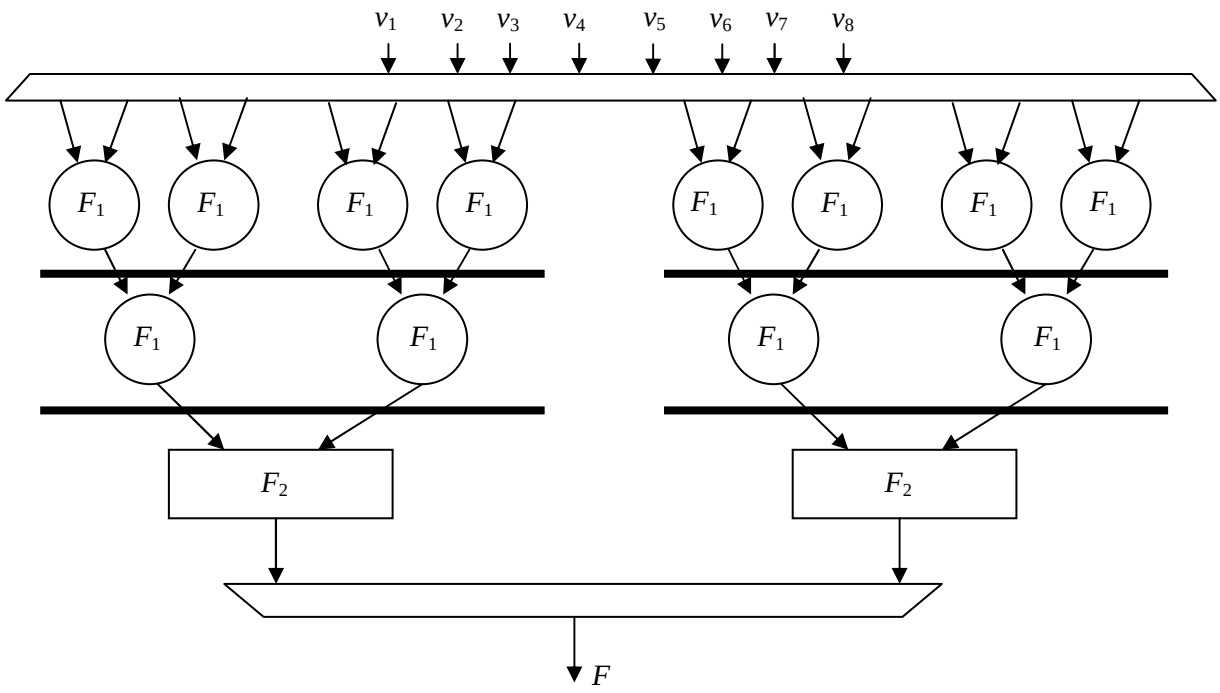


Figure 2-9: Full-parallel architecture ($\Gamma = 14$, $n_c = 1$, $n_s = 2$) for the computation of F

2.1.5.3 Architecture comparison

Comparing architectures with equivalent memory requirements can be done by comparing their complexity in terms of computational resources Γ. In this case, the ratio D_s / Γ is proportional to the activity, which is a sufficient parameter to evaluate the efficiency of the architecture. In fact, it contains all the information about the hardware complexity of the computational resources and their utilization in the execution of the algorithm.

The model we have derived does not take into account all the hardware parameters of each operator: number of bits, propagation delay, complexity, etc. A more comprehensive architectural model than the one developed in this section could be derived, taking notably into account the complexities of the elementary operators. However, we will se later on that our model is sufficient for our application.

2.2 State-of-the-art of turbo decoder implementation

After outlining the sequential decoding scheme that is usually used for turbo decoder implementations, we will briefly describe the fixed-point Max-Log-MAP algorithm. Then, we will recall the complexity results of this algorithm given in chapter 1.5, and our model of complexity will be validated using hardware synthesis results. Our fundamental model of efficiency for turbo decoder architectures will then be derived and used to compare the efficiency of the two schedules *SW-*Σ^- and *SW-*Σ^0 introduced in the previous chapter. This section ends with a description of practical hardware implementations of recursion units and of their characteristics.

2.2.1 Sequential decoding

Iterative decoding of a two-dimensional turbo code involves an exchange of extrinsic information between two SISO decoders, namely decoder 1 and decoder 2 for the natural and interleaved order, respectively. The execution of the decoder 1 and decoder 2 can be either sequential or concurrent as depicted in Figure 2-10 and Figure 2-11, respectively. For sequential decoding, the decoders are executed successively. The decoding process starts arbitrarily with either one decoder, decoder 1 for instance. After decoder 1 is completed, decoder 2 starts its processing, and so on. For the concurrent decoding, both decoders are executed concurrently. The extrinsic data are exchanged only once the iterations are completed. As such, this decoding scheme exhibits two sequential and independent processes that do not share any information together. Thus the number of iterations effectively performed is halved, and this technique is not attractive compared to the sequential decoding. For this reason, the sequential decoding will be considered in the sequel.

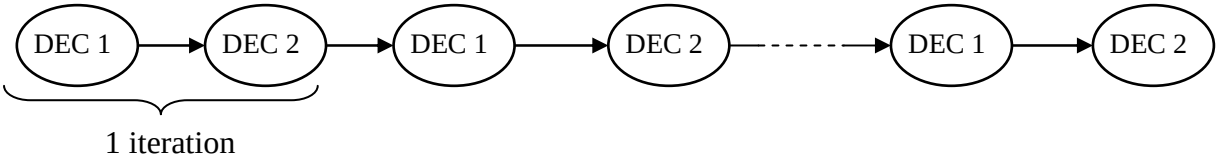


Figure 2-10: Sequential decoding of a two-dimensional turbo code

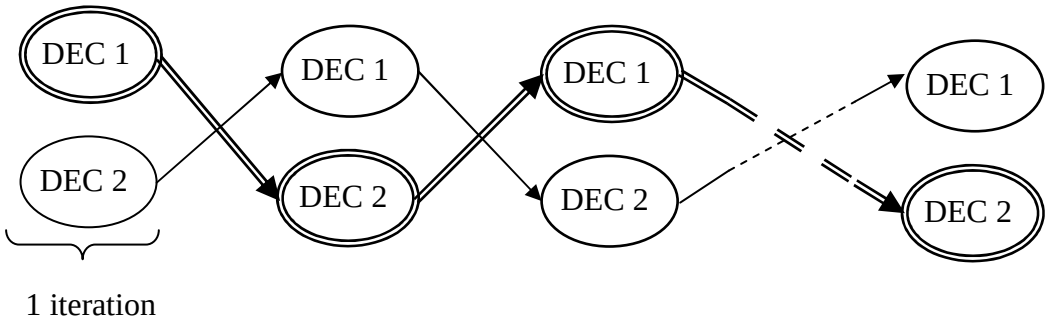


Figure 2-11: Concurrent decoding of a two-dimensional turbo code

2.2.2 Fixed-point Max-Log-MAP algorithm

A state-of-the-art turbo decoder implementation comprises SISO decoders implementing a fixed-point Max-Log-MAP algorithm. Fixed-point computation is mandatory for most architectures in hardware and software. The transformation of a floating-point representation to a fixed-point representation of the Max-Log-MAP algorithm has been widely addressed in the literature [Montorsi *et al.*-01a][Wu *et al.*-99][Masera *et al.*-99][Hsu *et al.*-99]. Some important results are briefly recalled hereafter.

The key parameter is the precision of the inputs of the decoder, *i.e.*. the number of bits w_d used to code the channel observations. This parameter influences the performance of the turbo decoder and also determines the internal size w_n of the state metrics.

The problem that arises when implementing a Viterbi recursion is the increase in the state metric values. A rescaling of the state metrics is therefore needed to bound their dynamic and avoid arithmetic overflows. This rescaling (or normalization) of the state metrics is also

necessary (although having a different justification) for numerical stability when formulating the MAP algorithm with probabilities, as has been shown in the first chapter.

This problem has been intensively studied for the Viterbi algorithm [Shung *et al*.-90]. In fact, it is well known that for this algorithm, the dynamic range of the state metric values corresponding to the same trellis stage (*i.e.*, the difference between the state metrics with the highest and lowest values) is bounded [Hekstra-89][Tortellier *et al*.-90]. This bound depends on the dynamic of the branch metrics and on the structure of the trellis and, more precisely, the constraint length of the code. Hence, the first rescaling technique subtracts the minimum state metric at each stage of the recursion. Since the relevant information is contained within the difference between state metrics, it remains unaltered. Yet, this rescaling scheme is computationally intensive since it requires the minimal state metric at each stage to be selected. An improvement of this technique compares the state metrics to a threshold, which is subtracted identically from all the state metrics only when all state metrics have exceeded it. This technique is particularly efficient when the threshold corresponds to a power of 2, chosen to be greater than the dynamic range of the state metrics. Using positive metrics, the comparison to the threshold results in a simple logical *NAND* between the MSBs of the state metrics. Then, a logical *AND* between the result and the MSBs enables the threshold to be subtracted, if required.

2.2.3 Complexity of one stage of the Max-Log-MAP algorithm

Let χ_u define the number of elementary operations involved in the production of one set of extrinsic outputs corresponding to one symbol of the *m*-binary CRSC code. The computational complexity has been assessed in the previous chapter and is recalled in Table 2-1. In this table, the respective complexities for the recursions and for the computation of the soft outputs are summarized. Assuming an equivalent complexity for addition and comparison-selection operations, the number of elementary operations (addition or comparison / selection) is given in the third column.

The RSC code considered here is the double-binary RSC code $v = 3$, $m = 2$ and $n = 4$ of rate 1/2. This code produces 2 redundancy bits for each symbol of 2 information bits. The computational complexities relative to the computation of the branch metrics, the recursion and the soft outputs are $\chi_{BM} = 32$, $\chi_{SM} = 56$ and $\chi_{SO} = 74$ elementary operations, respectively. Using the Max-Log-MAP algorithm, the total number of elementary operations χ_u for the production of the soft outputs for one output symbol is given by $\chi_u = 2\chi_{BM} + 2\chi_{SM} + \chi_{SO}$.

A more precise evaluation of the complexity of the computational units is given by the synthesis of the architecture considered on the respective circuits. The results are given in Table 2-2. The computational complexity is also recalled in the last column. For each module (state metric computation, branch metric computation or soft output computation), its fraction among the overall complexity is also given. The synthesis of the basic modules was performed for both FPGA and ASIC circuits assuming a quantization of the received samples on $w_d = 4$ bits. On FPGA, the complexities are given for circuits using 4-input LUTs in the Logic Elements: the devices Cyclone, Cyclone II and Stratix for Altera; Spartan3 and VirtexII

for Xilinx. The synthesis has been performed using QuartusII and Symplify for the Altera and Xilinx circuits, repectively. For an ASIC implementation, the synthesis results were obtained using Design Compiler from Synopsys and a standard library in a 0.18µ technology provided by ST Microelectronics (78 kgates/mm²).

	Addition	**Comparison-selection**	**Total elementary operations**
Branch metrics (forward or backward) χ_{BM}	$n_{ADD\_BRANCH}(n)+2^m$ where: $n_{ADD\_BRANCH}(n) = 4 \cdot C_n^2$	-	$n_{ADD\_BRANCH}(n)+2^m$
One step recursion (forward or backward) χ_{SM}	$2^{\nu+m}$	$2^{\nu}(2^m-1)$	$2^{\nu}(2^{m+1}-1)$
Soft outputs χ_{SO}	$2^m(2^{\nu}+3)-1$	$2^m2^{\nu}-1$	$2^m(2^{\nu+1}+3)-1$
Total computations for one symbol χ_u	$3.2^m(2^{\nu}+1)+ 4(n-1)+2^{m+1}-1 + n_{ADD\_BRANCH}(n)$	$2^{\nu}(3.2^m-2)-1$	$3.2^m(2^{\nu+1}+1)- 2^{\nu+1}+ 8(n-1)+2^{m+1}-2$

Table 2-1: Computational complexity of the Max-Log-MAP algorithm

Circuit	**FPGA complexity (LE)**		**ASIC complexity (gates)**	**Computational complexity**
Device or technology process	Altera Cyclone, Cyclone II Stratix	Xilinx VirtexII	ST 0.18µ	-
Branch metrics χ_{BM}	256 (8 %)	319 (10 %)	1365 (9%)	32 (13 %)
State metric χ_{SM}	740 (23.5%)	749 (23 %)	3345 (22 %)	56 (22 %)
Soft outputs χ_{SO}	1166 (37 %)	1145 (35 %)	5978 (38 %)	74 (30 %)
Total χ_u	3158	3281	15398	250

Table 2-2: Hardware complexity of the Max-Log-MAP algorithm for 4-input bits and RSC code with parameters (ν = 3, m = 2, n = 4)

These synthesis results demonstrate the validity of our model for evaluating the complexity of the algorithm. In particular, the relative complexities for each module are equivalent for the circuits considered. Therefore, in the following of the manuscript, the complexity of the Max-Log-MAP algorithm will be evaluated by counting the number of elementary operators. It will also be used to evaluate the activity of the turbo decoder architecture.

2.2.4 Throughput of a turbo decoder architecture

Let us derive a model for the throughput of a turbo decoder architecture, expressed in bits per second. For a turbo decoder corresponding to a 2-dimensional *m*-binary code, the number n_a of elementary operations to compute one output sample *(i.e. m* bits) is

$$n_a = \frac{2 \cdot I_{\max} \cdot \chi_u}{m}, \quad \textbf{(2-7)}$$

where I_{max} corresponds to the number of iterations performed by the turbo decoder, and where χ_u represents the computational complexity given by $\chi_u = 2\chi_{BM} + 2\chi_{SM} + \chi_{SO}$. The throughput is thus expressed in bits per second as

$$D_b = f_{clk} \cdot \alpha(\Gamma) \cdot m \cdot \frac{\Gamma}{2 \cdot I_{\max} \cdot \chi_u} \text{ [bits/s].} \quad \textbf{(2-8)}$$

2.2.5 Choice of the schedule

For a very high-speed implementation of the Max-Log-MAP algorithm, the maximum degree of parallelism of the algorithm is exploited, *i.e.* each operation of the algorithm is bound to a hardware operator. This results in a full-parallel architecture, no operator is reused to compute several operations at distinct phases of the algorithm execution. In order to obtain a high frequency, the architecture is pipelined and produces its outputs at each cycle after a pipeline latency equal to *l*. To implement the Max-Log-MAP algorithm, such a full-parallel architecture would require as many forward state metrics computation units as the number of trellis stages. The same holds for the backward state metrics computation units and soft output computation units. Such a full-parallel approach has been used in [Worm *et al*.-01] and results in a very high-throughput architecture: a new set of inputs corresponding to a whole trellis can be fed at each clock cycle, resulting in a throughput of 1 frame per cycle. This solution leads to a large pipeline depth, which requires a large number of hardware resources that cannot scale with practical implementations.

Consequently, a locally full-parallel architecture is used for the implementation of the Max-Log-MAP algorithm. A locally full-parallel architecture instantiates all the resources that are needed for the computations of a single trellis stage (forward recursion, backward recursion and computation of the soft outputs). A straightforward implementation of such a locally full-parallel architecture instantiates $\Gamma_{SW-\Sigma^-} = 2 \cdot \chi_{BM} + 2 \cdot \chi_{SM} + \chi_{SO} = \chi_u$ elementary operators. Each of the χ_u operators is used once for the computation of one set of soft outputs. The allocation of the operations of the algorithm to the available resources can be either with schedules Σ^- and Σ^+ or with their sliding window equivalents *SW*-Σ^- and *SW*-Σ^+. Schedule Σ^0 cannot be used, because it requires the computation of two sets of soft outputs simultaneously.

Schedule *SW*-Σ^-:

The assignment of the operations of the Max-Log-MAP algorithm to the resources of the architecture according to schedule *SW*-Σ^- is presented in Figure 2-12. The recursions are initialized with the NII technique (see chapter 1.5). For the sake of clarity of the figure, the pipeline latency of *l* cycles is allocated to the end of the data path, *i.e.* at the computation of the soft outputs (SOU). However, in the general case, the pipeline registers are uniformly distributed along the data path. It is assumed that the observations and extrinsic data are stored in the observation and extrinsic memories, respectively, and are available at time $\tau = 0$.

These memories are external to the SISO decoder. The backward recursion precedes the forward recursion and the backward state metrics are stored in a memory. They are read out from the memory during the forward recursion to be combined to the forward state metrics to produce the corresponding soft outputs. In this architecture, the SISO decoder inputs (observations and extrinsic data) that are used during the backward recursion to compute the branch metrics are stored in a local memory and reused during the forward recursion. This saves a second read access to the external memories. This illustrates the third law given above, since communications to the external memories (requiring interconnections) is exchanged for local memorization.[1]

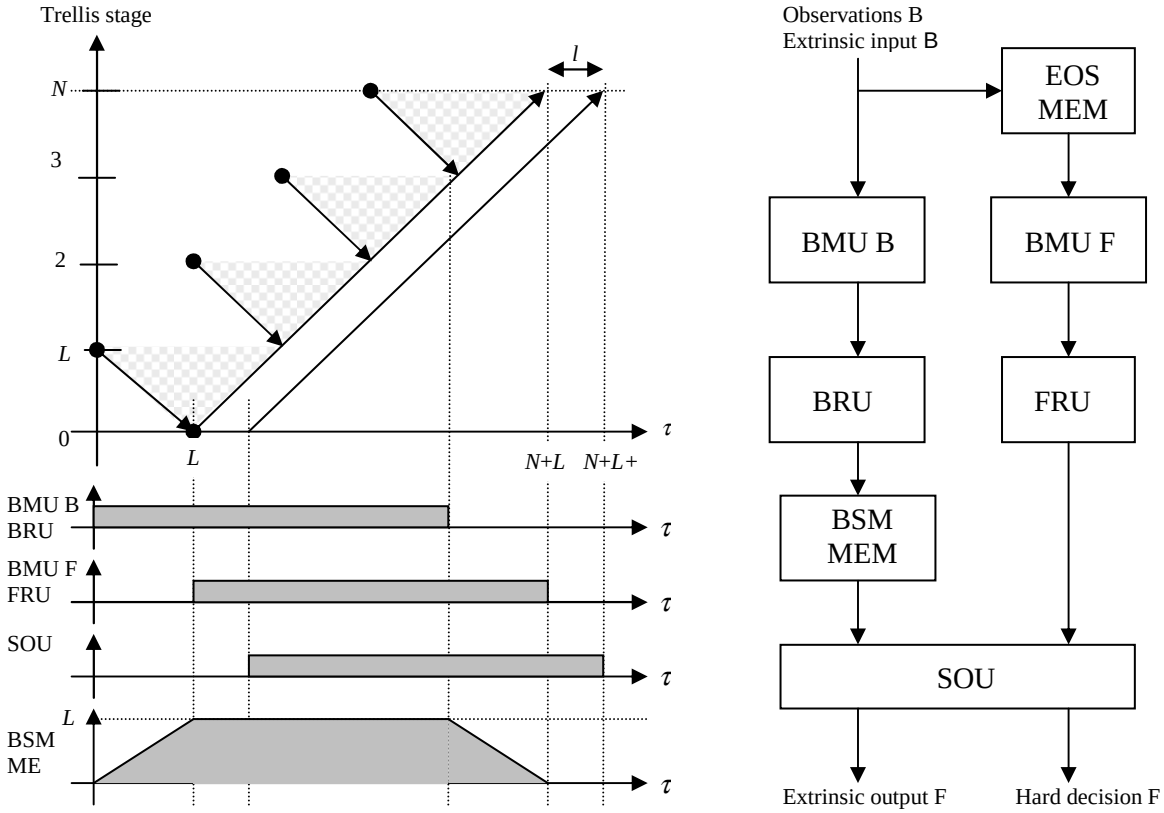


Figure 2-12: Allocation of the resources to the execution of the Max-Log-MAP algorithm with schedule *SW- Σ⁻* and corresponding architecture: block diagram (right), algorithm schedule (top left) and activities of computation units and memory usage (bottom left)

The corresponding locally full-parallel architecture presented in Figure 2-12 comprises two Last In - First Out local memories (LIFO), which inverse the order of the data:

- a Extrinsic and Observation Sample memory (EOS MEM) that stores the inputs of the SISO read during the backward recursion over each window. The data is read back (in a reversed order) during the forward recursion performed on the same window.

[1] In fact, the local memorization of the input data can be optimized according to the first law: the computations of two sets of branch metrics can be simplified by modifying the position and the size of the local memory. This optimization will be addressed in chapter 6.

- a backward state metrics memory (BSM MEM) that stores the backward states metrics produces during the backward recursion over a window length. These data are read back by the forward recursion.

Further, it comprises five computation units:

- two branch metric units (BMU F and BMU B) that compute the branch metrics for the forward and backward recursions, respectively, using the inputs of the decoder: the observations and the extrinsic data.
- a backward recursion unit (BRU) that uses the branch metrics produced by BMU B to compute for each window the backward states metric that are stored in BSM MEM.
- a forward recursion unit (FRU) that computes the forward state metrics using the branch metrics computed by BMU F.
- soft output unit (SOU) that computes the soft outputs using the forward state metrics and the backward state metrics read out from the state metric memory BSM MEM.

The two recursion units work simultaneously and exchange data through the memories EOS MEM and BSM MEM: the forward recursion unit works on the current window while the backward recursion unit works on the next window.

Since, with the schedule *SW-Σ⁻*, each operator is used N times during $N+L+l$ cycles, its activity is given by

$$\alpha_{SW-\Sigma^-} = \frac{N}{N+L+l}. \qquad \textbf{(2-9)}$$

Due to the term L in the denominator, the activity decreases for small frame sizes. On the contrary, when the size of the frame tends toward infinity, the activity tends toward 1. Assuming that the same component decoder performs all the iterations, the throughput of the decoder is given by:

$$D_b = f_{clk} \cdot \frac{N}{N+L+l} \cdot \frac{m}{2 \cdot I_{\max}} \text{ [bits/s].} \qquad \textbf{(2-10)}$$

Example 2-5:

For a double-binary RSC turbo code with parameters ($\nu = 3$, $m = 2$, $n = 4$) is decoded using a Max-Log-MAP algorithm with a schedule *SW-Σ⁻*. For f_{clk} = 100 MHz, I_{max} = 5, L = 32, l = 8, and N = 212, the activity equals $\alpha_{SW-\Sigma^-}$ = 0.84 and the throughput amounts to D_b = 16.8 Mbits/s.

Schedule *SW-Σ⁰*:

Using schedule *SW-* Σ^0 instead of schedule *SW-Σ⁻* is a straightforward solution to increase the throughput of the turbo decoder architecture. For this schedule, the computation of the soft outputs is performed simultaneously during the second half of each forward and backward window recursions. Likewise, the computation of the branch metrics is performed concurrently during the computations of the first half of the forward and backward window

recursions. In this architecture, the data read from the observation and extrinsic memories during the forward recursion (or backward recursion) are stored in a memory and reused during the backward recursion (or forward recursion respectively). Hence, the architecture requires $\Gamma_{SW-\Sigma^0} = 2\cdot\chi_{SM} + 2\cdot\chi_{SO} + 2\cdot\chi_{BM}$ elementary operators. The allocation of the operations of the algorithm to the resources of the architecture according to schedule *SW-*Σ^0 is described in Figure 2-13.

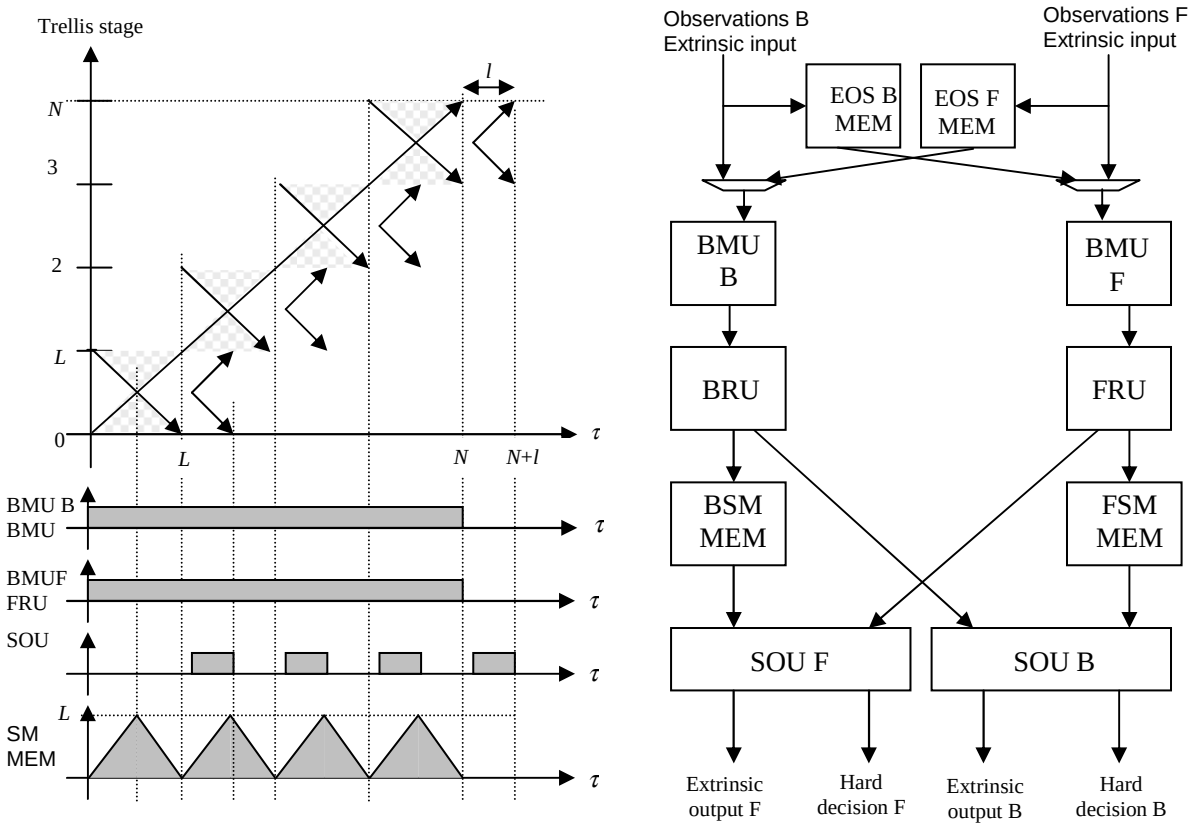


Figure 2-13: Allocation of the resources to the execution of the Max-Log-MAP algorithm with schedule *SW-* Σ^0 and corresponding architecture

It is worth noting that, for this schedule, the soft output computation units and the branch metric computation units are used less than half of the time. The under-utilization of these resources has a significant impact on the associated activity. Let $\beta_{SW-\Sigma^0}$ represent the ratio of computational complexity χ_u over the available computational complexity of the architecture $\Gamma_{SW-\Sigma^0}$, $\beta_{SW-\Sigma^0} = \chi_u / \Gamma_{SW-\Sigma^0}$. The activity is then expressed as

$$\alpha_{SW-\Sigma^0} = \beta_{SW-\Sigma^0} \cdot \frac{N}{N+l}. \qquad \textbf{(2-11)}$$

The second term of this product shows that the activity of *SW-*Σ^0 is independent of the window length *L*. Thus the throughput loss for small frames is reduced compared to *SW-*Σ^-. Unfortunately, due to the first term $\beta_{SW-\Sigma^0}$, the architecture is rather inefficient, even for long frames: when the size of the frame tends towards infinity, the activity tends toward this first

term, while it tended toward 1 for schedule *SW-Σ⁻*. Nevertheless, the throughput is increased to

$$D_b = f_{clk} \cdot \frac{N}{N+l} \cdot \frac{m}{2 \cdot I_{\max}} \text{ [bits/s].} \quad \textbf{(2-12)}$$

Taking the computational complexities of section 2.2.3, the term $\beta_{SW-\Sigma^0}$ in the activity can be computed using the following relations:

$$\chi_u = 2\chi_{BM} + 2\chi_{SM} + \chi_{SO} \quad \textbf{(2-13)}$$

$$\Gamma_{SW-\Sigma^0} = 2\chi_{BM} + 2\chi_{SM} + 2\chi_{SO} \quad \textbf{(2-14)}$$

Therefore the first term in the activity amounts to $\beta_{SW-\Sigma^0}$ = 0.77, which is appromixated to 0.75. This maximal activity is reached for long frame sizes. The next example evaluates the activity reduction for small frame sizes.

Example 2-6:

For the same code as in Example 2-5, the activity is reduced to 0.67, which is lower than 0.84 obtained for *SW-Σ⁻*. But the throughput is increased from 16.8 Mbits/s to 19.3 Mbits/s.

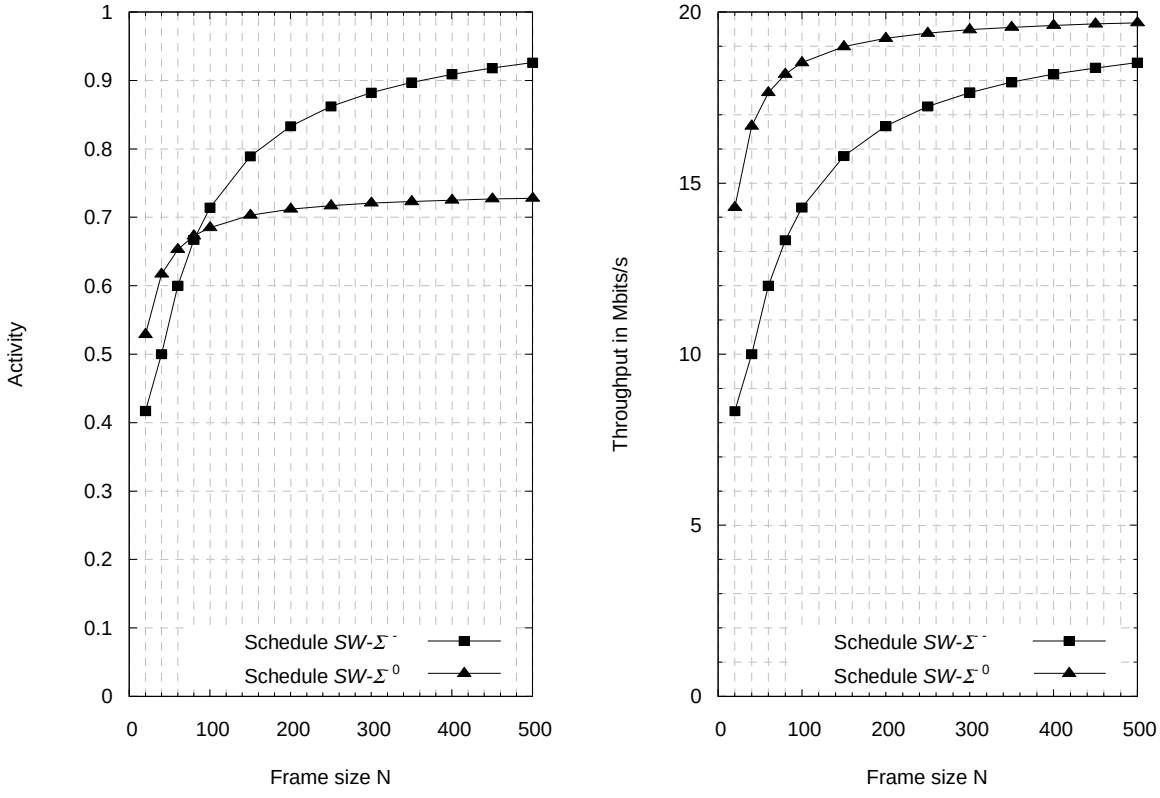


Figure 2-14: Comparison of the activities and throughput of the architectures for schedules *SW-Σ⁻* and *SW-Σ⁰* as a function of the size of the frame *N*, for code 2/3, *L*=32, *l* = 8, f_{clk} = 100 MHz, I_{max}= 5

A comparison of the activities and throughput of the two schedules and their corresponding architectures is presented in Figure 2-14 for the double-binary turbo code. From this comparison, for frame sizes greater than 100 symbols schedule *SW-Σ⁻* represents the best compromise in terms of activity, which is, as we have seen, the appropriate metric for

comparing two different architectures. This conclusion would not have appeared so clearly if we had focused only on throughput and complexity separately.

2.2.6 Implementation of recursion units

Each recursion unit (forward or backward) is implemented according to the description of Figure 2-15. At each processing stage, the current set of state metrics is stored in a register called state register controlled by the clock *clk*. This register can also be initialized to an external set of state metrics through the input multiplexer, used at each window start. The next set of state metrics is computed by 2^{ν} Add-Compare-Select (ACS) operations, one for each state. Each ACS operation, involves additions of the branch metrics to the corresponding current state metric and selection of the minimum metric among the 2^{m} concurrent metrics. For $m \geq 2$, the selection over the 2^{m} concurrent metrics is performed by a tree of 2^{m-1} comparison-selection operation distributed over $\log_2(2^{m})$ stages. The set of current state metrics is connected to the computation units through a shuffle network, which permutes the state metrics according to the trellis connections. This shuffle network is different for the forward and backward recursion units. After the ACS operation, the resulting state metrics are rescaled using dedicated logic.

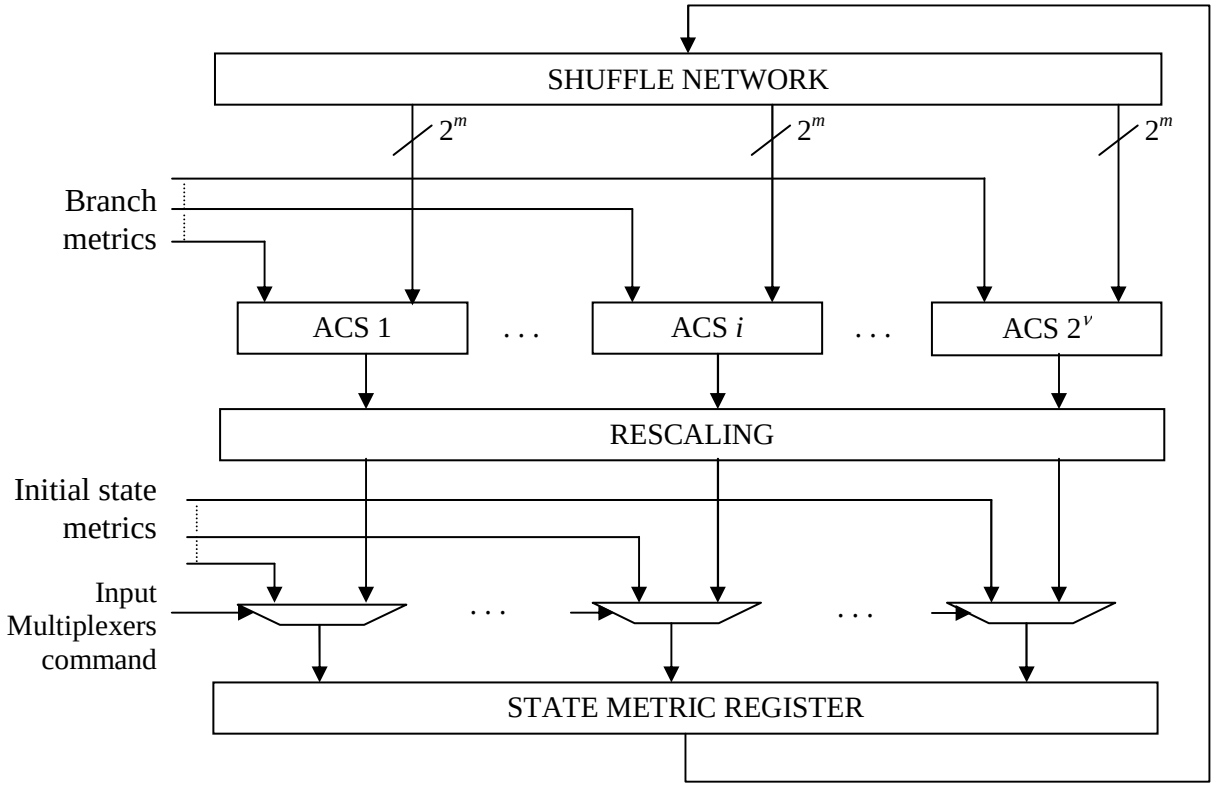


Figure 2-15: Architecture for the computation of a forward or backward recursion

2.2.7 Maximal working frequency and pipelining

Generally, the critical path for an efficient turbo decoder architecture corresponds to the path in the feedback loop of the recursion units. It is composed of the propagation along the addition, the multi-stage comparison-selection operation, the rescaling and the input multiplexer. Such an architecture comprises several pipeline registers in the other units in

order to achieve a maximum frequency of at least the frequency of the recursion unit. The latency of l cycles described in Figure 2-12 takes into account these pipeline registers, and also the latencies associated with the read and write operations to the memories (extrinsic memory, state metric memory, etc).

2.3 Increasing the throughput of turbo decoders

In this section, several techniques to increase the throughput of a turbo decoder are discussed. They are derived from the relation obtained in section 2.2.4 and recalled below.

$$D_b = f_{clk} \cdot \alpha(\Gamma) \cdot m \cdot \frac{\Gamma}{2 \cdot I_{\max} \cdot \chi_u}. \quad \textbf{(2-15)}$$

Relation (2-15) is the **fundamental relation** driving the different directions of our research towards an efficient high throughput architecture for iterative turbo decoding. Each term of this relation initiates a complete research field. The techniques described in the next sub-sections derive from the following directions for improvement:

- choosing the algorithm that minimizes the number of elementary operations χ_u (section 2.3.1).
- decreasing the number of iterations I_{max} either dynamically or statically (section 2.3.2)
- increasing the number of resources Γ by means of parallelism (section 2.3.3).
- improving the utilization of the resources by increasing the activity α (section 2.3.4).
- maximizing the clock frequency f_{clk} of the architecture (section 2.3.5).

Improving the efficiency of the architecture can also be achieved by hardware design techniques enabling a reduction in the circuit area. This is discussed in section 2.3.6.

2.3.1 Choice of the decoding algorithm of the SISO decoder.

The objective of the choice of a schedule is to select the algorithm with the best performance/complexity trade-off. It is worth noting that a small modification of the decoding algorithm can significantly decrease the complexity of its implementation. It has been shown, for instance, in the first chapter that omitting the correction term in the Log-MAP algorithm reduces the complexity of each comparison-selection operation by saving an addition and a LUT. Moreover, for a double-binary turbo code, the degradation with the Max-Log-MAP algorithm is less than 0.1 dB. For this reason, the Max-Log-MAP algorithm is chosen in our work.

In addition, in iterative decoding, a loss of performance (BER) induced by a degradation of the component decoder can be partially recovered by performing more decoding iterations. For instance, the simplification of the Log-MAP algorithm to the Max-Log-MAP algorithm illustrates this concept. When working in an area where additional iterations improve the BER, performing more iterations with the sub-optimal algorithm can yield a better BER than the optimal algorithm. Note however that asymptotically (at a high number of iterations) the

optimal algorithm always performs better. This example illustrates the fundamental trade-off in iterative decoding between the number of iterations and the complexity. It leads to the fourth architectural law for iterative decoding (see 2.1.5.1 for the other laws):

- Law 4: Algorithm complexity and number of iterations can be exchanged (up to a certain extent).

2.3.2 Decreasing the number of iterations.

Generally, the iterative process of a turbo decoder is stopped after a predetermined maximal number of iterations I_{max}. Hence, decreasing the number of iteration can be achieved either statically by fixing a lower maximum number of iterations I_{max}, or dynamically by using a stopping criterion, which enables the decoder to stop the iterations once the criterion is satisfied.

2.3.2.1 Static reduction in the number of iterations

In the first step, we study the reducing the maximum number of iterations I_{max}. Obviously, this very simple technique comes with a degradation of performance. However, the performance of iterative decoding is undeniably less sensitive to a reduction in the number of iterations at the end of convergence of the iterative process than at the beginning. It is well known that the gain in error decoding performance obtained by a decoding iteration decreases with the number of iterations. As a consequence, a reduction of a few iterations at the end of the convergence of the iterative process enables a significant increase in throughput at a reduced cost in performance degradation. This technique of reduction in I_{max} is particularly important, since it is a very flexible solution, requiring neither additional complexity nor design effort. This remark will be crucial later on, especially in chapter 4. In fact, this simple technique will serve as a benchmark for comparing the efficiency of the other techniques that will be proposed for augmenting the throughput of the decoder and that might induce performance degradation.

2.3.2.2 Dynamic reduction in the number of iterations

Generally, not all iterations are always necessary since all the errors might be corrected after the first I ($I < I_{max}$) decoding iterations. Assuming a method (stopping criterion) to detect the absence of remaining errors with a high probability then it is possible to stop the iterative process. This dynamic (data-dependent) reduction in the number of iteration enables either a large increase in the average decoding throughput or an improvement in performance. Different stopping criteria can be envisaged either external or internal to the turbo decoder. An external stopping criterion could for instance be a Cyclic Redundant Check (CRC) that is added to the information message [Shibutani *et al*.-99]. Such a criterion is very efficient, since the iterative process can be stopped once the frame has been corrected. However, the transmission of the redundancy associated with the CRC results in a reduction in the overall code rate of the system that can be considerable for small frame sizes. In addition, such an external stopping criterion is not always included in existing systems. These two reasons explain the interest for internal stopping criteria, which are based either on the hard decisions

or on soft decisions. Several criteria have been proposed in [Hagenauer *et al*.-96], [Berrou *et al*.-97], [Shao *et al*.-99] and [Wu *et al*.-00]. They all involve evaluating the convergence of the iterative decoding process. This latter is stopped once an ad-hoc metric value exceeds a given threshold.

Application:

For most of the practical applications of turbo decoding, a minimal throughput has to be guaranteed. When the dynamic stopping method is applied on a frame-by-frame basis, it cannot increase the minimal throughput that corresponds to the worst case of I_{max} iterations, without performance degradation. The only advantage of this technique is to decrease significantly the power consumption of the decoder.

In order to increase the minimal throughput of a turbo decoder architecture, stopping rules applied to several frames have to be considered. This technique requires buffering of the input frames and distributing the computational power among several consecutive frames. It enables to approach the throughput obtained with the average number of iterations. The design of such systems has to take into account the increase in latency (more precisely latency jitter), the memory resources associated with the buffers, the throughput and the performance of the decoder [Martinez *et al*.-03].

2.3.3 Increasing the computational resources

The natural way to increase the throughput of an architecture is to increase the computational resources. The objective for an efficient architecture is to linearly increase the computational complexity of the decoder for a linear increase in the throughput. The first architecture, the serial architecture (each iteration dedicated to a distinct decoder), that was proposed in [Berrou *et al*.-96] did not achieve this objective. Note, however, that for this code, this architecture was particularly well suited to the code considered, since the turbo coded transmission was not on a frame-by-frame basis, but continuous. Terminated turbo code opens new architectural possibilities and in particular parallel architectures.

2.3.3.1 Serial architecture

In the serial architecture depicted in Figure 2-16, a cascade of several component SISO decoders separated by memories performs the successive iterations of turbo decoding. The memories between the SISO decoders are required to implement the interleaving and de-interleaving functions. This duplication of the memories, once for each half-iteration, leads to a considerable increase in the area of the decoder, especially for very long frame sizes.

This serial architecture was used in the first commercial implementation of a turbo decoder [CAS-95], built around five 8-state soft output Viterbi decoders, thus enabling 2.5 iterations. It also implements four memories for interleaving and de-interleaving a frame of size 1024 bits. A serial architecture was also suggested in the implementation of the *TURBO4* chip [Jézéquel *et al*.-97], which implements two SOVA decoders for a 16-state binary trellis and corresponding memories for interleaving and de-interleaving a frame size of 2048 bits. Since each *TURBO4* chip performs a single iteration, a cascade of I_{max} modules enables I_{max}

iterations to be performed. Even if these first circuits where limited in interleaver size and suffered from the duplication of the corresponding memories, they demonstrated that practical implementations of turbo decoders were possible. It brought confidence in the turbo technology that doubtlessly shortened the delay for their integration into communications standards. Only six years after their invention, they were adopted by the CCSDS for deep-space communications [CCSDS-99].

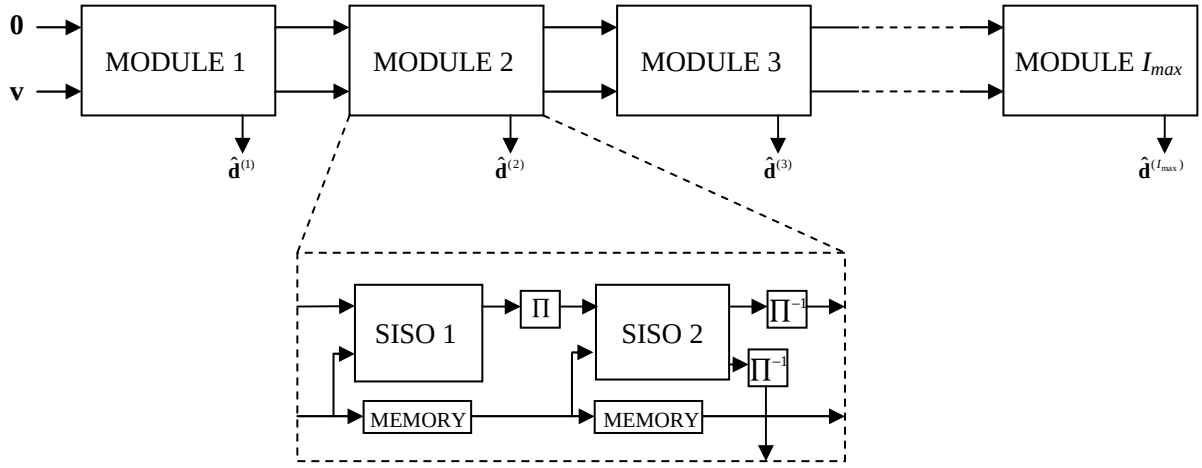


Figure 2-16: Serial architecture for real-time turbo-decoding

For the serial architecture, the throughput increases linearly with the area including the memories and the computational units. The objective of the architecture presented in the next section is to achieve a linear increase in the throughput with the area of the computational units only.

2.3.3.2 Parallel architecture

To overcome the duplication of memories in a serial architecture, while maintaining the same throughput level as offered by a serial architecture, several SISO decoders working in parallel are used to decode a single frame. To this end, the Max-Log-MAP algorithm has to exhibit more parallelism than the state-of-the-art implementation, particularly in the forward and backward recursions. For the implementation described in section 2.2.5, only one forward recursion and one backward recursion are performed simultaneously. In fact, thanks to the sliding window technique, this implementation has a higher degree of parallelism than the conventional Max-Log-MAP algorithm, for which the two recursions are sequential. Other sliding window techniques process more than two recursions simultaneously [Boutillon *et al*.-03][Dingninou-01], but they are intended to reduce the memory requirements. The throughput remains unchanged. These schemes use the first architectural law and exchange memorization of state metrics against computation of state metrics, thus adding additional simultaneous recursions.

In order to exhibit more parallelism in the Max-Log-MAP algorithm, the trellis corresponding to the encoded frame is divided into a number of trellis sections, as described for the first time in [Hsu *et al*.-98]. A trellis section is defined as a set of consecutive trellis stages. These trellis sections can then be processed independently and simultaneously by

multiple recursion and soft output computation units. This is made possible by the sliding window principle. According to this approach, state metrics calculation can be started at an arbitrary position in the trellis. The computation units processing the same trellis section are grouped into an entity called a processor. For P processors working on P distinct trellis sections of size $M = N / P$, the objective is that the throughput of the architecture is P times the throughput of a single processor. In order to achieve this objective of linear increase in the throughput with computational complexity, three issues have to be addressed. Before giving more details about these issues, let us first illustrate the benefits of the parallel architecture over the serial one.

To this end, the area of both solutions are compared for an ASIC implementation using a standard cell library in 0.13µ provided by TSMC[1] (around 200 kgates/mm²). The processor implementing a Max-Log-MAP algorithm according to schedule *SW-Σ⁻* (see Figure 2-12) for a double-binary RSC code (8,2,4) has been synthesized. It achieved a maximal working frequency of f_{clk} = 200 MHz for an area A_P = 0.15 mm². The complexities of the observation and extrinsic information memories amounts to A_I = 0.67 mm² (simple port) and A_E = 0.59 mm² (double-port), respectively. For a throughput of 200 Mbits/s at 8 iterations, P = 8 processors are required (at each clock cycle, each processor produces m = 2 extrinsic data). For the serial architecture, the memories are duplicated accordingly and the area is thus given by $A_{TD}^{S} = P(A_P + A_E + A_I)$. For the parallel architecture, the terms related to the memories can be extracted from the parentheses. An additional observation buffer is required to store the next incoming frame while decoding the current one. The area is then reduced to $A_{TD}^{P} = 2A_I + A_E + P \cdot A_P$. Table 2-3 presents the comparison of the two solutions for a frame size of N = 4096 duo-binary symbols. Because of memory duplication, which represents more than 90% of the total area, the classical serial architecture is almost four times as complex as the parallel architecture. The gap is even more important with longer block size or more iterations.

	A_P **0.15 mm²**	A_I **0.67 mm²**	A_E **0.59 mm²**	A_{TD}
Serial architecture	1.20 mm² (8)	5.38 mm² (8)	4.72 mm² (8)	11.3 mm²
Parallel architecture	1.20 mm² (8)	1.34 mm² (2)	0.59 mm² (1)	3.13 mm²

Table 2-3: Area comparison of the serial and parallel architectures for I = 8, f_{clk} = 200 MHz, N = 4096 (the number of units is given in parentheses)

In addition to the reduction in complexity, parallel decoding also enables us to divide the decoding delay by a factor P.

2.3.3.3 Recursion initialization with parallel architectures

The initialization of recursions can be handled by using the sliding window approach (see chapter 1.5). The first technique addressed involves pre-processing steps in order to initialize the recursions. This solution has been used in [Wang *et al.*-01], [Wang *et al.*-02] and [Dobkin

[1] Taïwan Semiconductor Manufacturing Company.

et al.-02] where the trellis is divided into overlapping trellis sections, on which the pre-processing steps are performed. In [Worm *et al*.-01], another parallel scheme using non-overlapping trellis sections is used, where pre-processing steps also initialize the recursions. Due to the pre-processing steps, these solutions increase the computational complexity of the parallel decoding algorithm. Similarly to the sliding window schedules, the recursions can be initialized in an iterative context by the corresponding final values obtained at the previous iteration [Giulietti *et al*.-02b][Yoon *et al*.-02]. In this case, the trellis is split into non-overlapping trellis sections.

Depending on the size of the trellis section, several schedules can be envisaged for decoding each trellis section: Σ^+, Σ^-, Σ^0. Each trellis section can be divided into smaller units (windows) of size L, and a sliding window schedule (SW-Σ^+, SW-Σ^- or SW-Σ^0) is performed on each trellis section. Figure 2-17 depicts such a parallel schedule using the SW-Σ^- schedule for all the trellis sections. The trellis is divided into $P = 4$ trellis sections of size M and each trellis section is decoded with a sliding window algorithm comprising $M/L = 3$ windows. The dotted arrows represent the initialization of the trellis section by the final values obtained at the previous iteration.

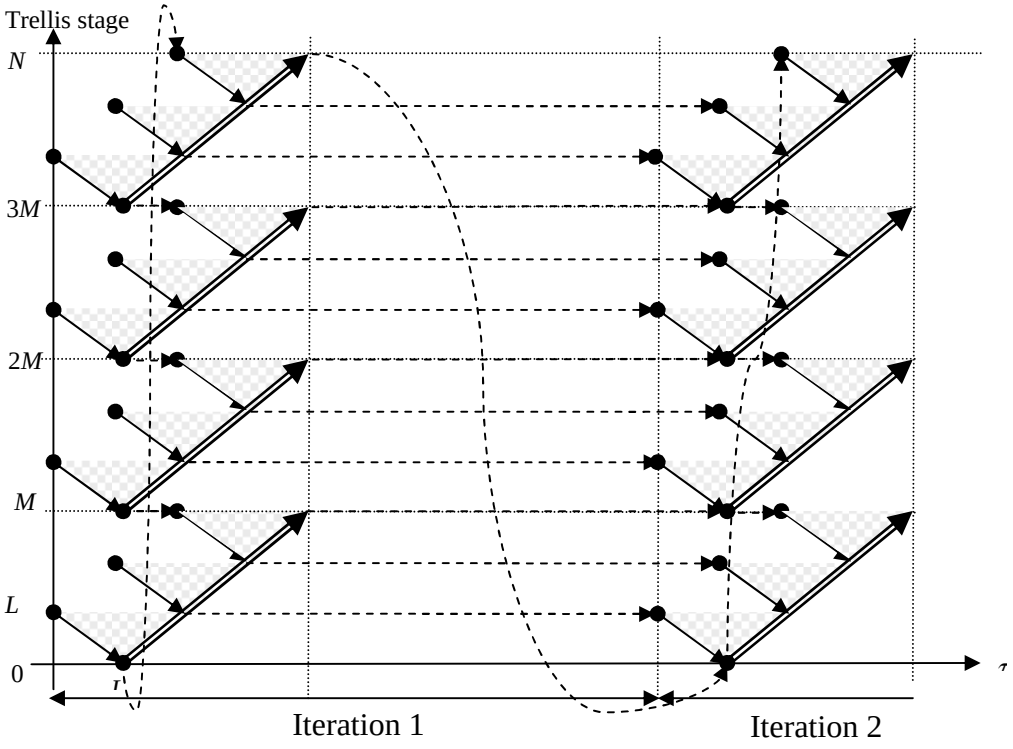


Figure 2-17: Parallel decoding with schedule SW-Σ^- and Next Iteration Initialization technique

As shown in chapter 1.5 this initialization technique does not degrade the performance. In fact, the fourth architectural law discussed in section 2.3.1 applies. The parallel sliding window algorithm does not differ significantly from the conventional non-parallel sliding window algorithm. If the size of the window is the same, the backward recursions produce state metrics of equivalent reliabilities. The only modification lies in the forward recursion

that is split into *P* independent recursions. The impact on the BER will be evaluated in chapter 6.

2.3.3.4 Iterative parallel decoding of turbo codes

The second issue with parallel decoding results from the encoding scheme of turbo codes including an interleaver for scrambling the information sequence. This issue is highlighted in this subsection and solutions are given to resolve it.

Parallel architecture:

Let us assume that the observations and the extrinsic data are stored in an observation memory and an extrinsic memory, respectively. During the iterative process the extrinsic information produced by one component decoder is used as *a priori* information by the other decoder. In the sequel, the *a priori* information denomination will not be used, but replaced by the unique denomination of extrinsic information. It is denoted **Z**. Thus, extrinsic information refers either to an extrinsic input read by the processor or to an extrinsic output produced by the processor. For sake of simplicity, two single-port memories storing the data in the natural and interleaved order can be used for the extrinsic information. This duplication of memory is not necessary and a single dual-port extrinsic memory can be used, which contains successively the extrinsic information produced by the natural and interleaved order. The corresponding architecture is described in Figure 2-18.

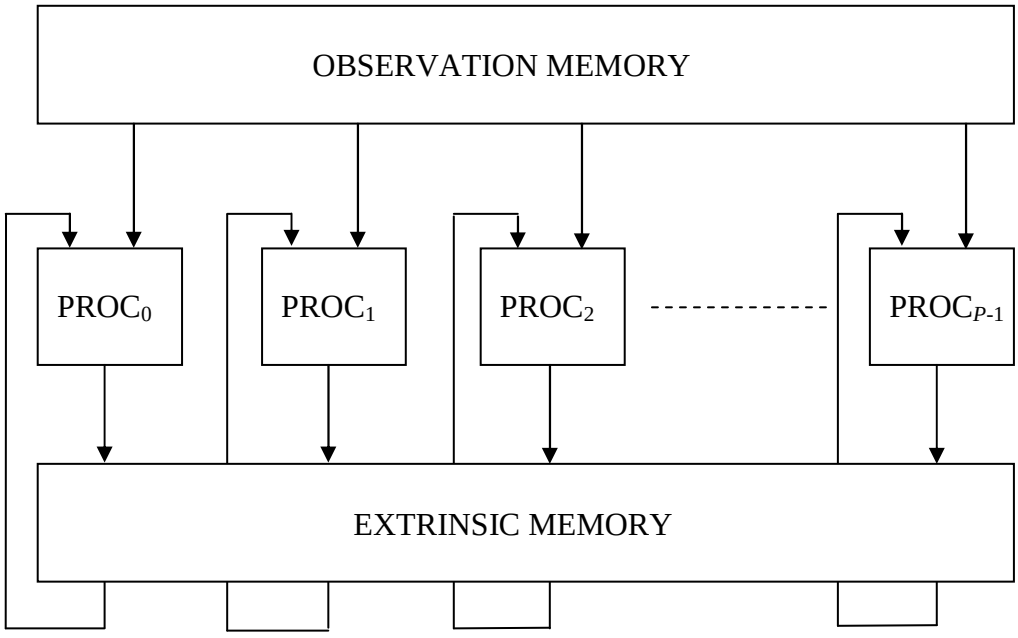


Figure 2-18: Parallel architecture for turbo decoding

The data stored in the memories are read by the processors according to the schedule of the algorithm. Similarly, the extrinsic information data are written back to the memory once they are produced by the processors. In this case, the extrinsic output produced by the processor is an update of the extrinsic input, and we assume that it is written into the same memory cell in which it was read. In this case, the unique extrinsic memory stores the extrinsic values $\mathbf{Z}_k$, $k = 0,\ldots, N\text{-}1$ in the natural order for instance.

Memory banks:

Since every processor performs the same decoding algorithm with the same schedule (on different trellis sections), they all access the memory at the same instants of time. Assuming that each processor performs a maximum of m_a accesses per cycle (read or write) to the memory, parallel decoding with P processors requires a maximum bandwidth of $P\cdot m_a$ accesses per cycle. The implementation of a memory on a hardware circuit has been described in section 2.1.2. It showed that a memory bank has a limited number of ports m_p. Consequently, to satisfy the bandwidth of $P\cdot m_a$ accesses per cycle required by parallel turbo decoding, the observation and extrinsic memories need to be separated into several memory banks. Using memory banks with m_p ports each, the minimum number of memory banks is $P\cdot m_a / m_p$. For instance, for dual-port memory banks, and $m_a = 2$ accesses per processor per cycle, at least P memory banks are required.

For the schedule described so far, the maximal number of accesses is $m_a = 2$. We assume an internal memorization (inside the processor) of the read data from observation and extrinsic memories (to be used by both forward and backward recursions) as described in section 2.2.5. Then, the number m_a of accesses to the extrinsic memory is equal to 2: one read access and one write access for schedules $SW\text{-}\Sigma^{-}$, two read accesses or two write accesses for schedule SW-Σ^{0}. On the other hand, only read accesses are performed to the observation memory. Hence, in the sequel, we restrict ourselves to the study of a single type of access (read or write) performed simultaneously by the P processors, one access per processor per cycle. P memory banks with one dedicated port are required for these accesses. The second access for each processor is served by a second dedicated port of the memory bank.

Memory organization:

This separation of the observation and extrinsic memories into several memory banks induces the need for a memory organization allowing parallel decoding. Finding a memory organization involves specifying the location (or mapping) of each data into one of the memory banks. This mapping has to ensure that at each time instant of the decoding process, $P\cdot m_a$ concurrent accesses to the memory bank can be performed, *i.e.* each access uses a different port among the $P\cdot m_p$ ports of the memory banks. For instance, with dual-port memory banks comprising one read port and one write port, P concurrent read accesses and P concurrent write accesses can be performed if the data are stored in P distinct memory banks.

In the context of parallel decoding of turbo codes, observation and extrinsic data are accessed in two different orders: read and write accesses in the natural order for the first dimension, read and write accesses in the interleaved order specified by the interleaver Π for the second dimension. Therefore, choosing an appropriate mapping function for decoding the turbo code in the natural order can lead to collisions in the accesses when decoding in the interleaved order. A collision occurs when two processors (or more) need to access the same memory bank at the same time instant, as depicted in Figure 2-19.

These collisions due to parallel decoding by multiple processors are called parallel conflicts. They are caused by the presence of the interleaver in the coding scheme. The

existence of parallel conflicts also depends on the schedule of parallel decoding and on the choice of the memory organization associated with parallel decoding. They represent a major bottleneck in high speed decoding of turbo codes. A straightforward workaround involves transforming each group of *g* conflicting accesses into *g* successive accesses, thus serializing the conflicting accesses. The obvious consequence is that it spoils the parallelism. State-of-the-art solutions to overcome parallel conflicts, along with ours, will be described in chapter 3.

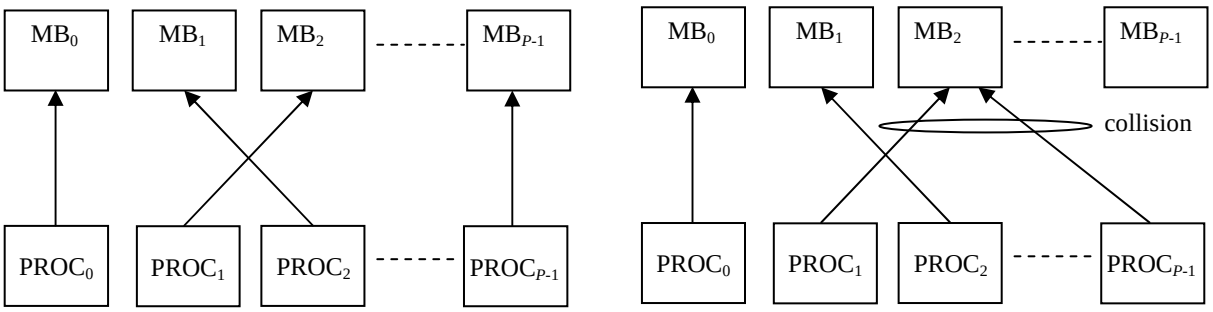


Figure 2-19: Absence / occurrence of collisions in the accesses to the memory banks

2.3.3.5 Resulting reduction in activity

As has already been stated, the activity drops for small frame sizes. The execution of the Max-Log-MAP algorithm according to schedule Σ has an activity given by the following relation

$$\alpha_{\Sigma} = \beta_{\Sigma} \cdot \frac{N}{N + \theta_{\Sigma} + l}. \qquad \textbf{(2-16)}$$

where $\theta_{\Sigma} = 0$ for schedule *SW*-Σ^0 and $\theta_{\Sigma} = L$ for schedules *SW*-Σ^-, and where $\beta_{\Sigma} = 0.75$ (see 2.2.5) for schedule *SW*-Σ^0 and $\beta_{\Sigma} = 1$ for schedules *SW*-Σ^-. In the context of parallel decoding, the size *M* of the trellis that is decoded by each processor decreases as the number of processors *P* increases. The activity is then obtained from (2-16) by replacing *N* by *M* = *N/P*:

$$\alpha_{\Sigma} = \beta_{\Sigma} \cdot \frac{N/P}{N/P + \theta_{\Sigma} + l}. \qquad \textbf{(2-17)}$$

For example, for *N* = 1024, *P* = 16 and *L* = 32, the activity of schedule *SW*-Σ^- is as low as 0.67, *i.e.*, the processors are idle one third of the total frame decoding time, which represents a dramatic loss of resource usage.

In conclusion, due to a reduction in activity in the context of parallel decoding, a linear increase in the throughput cannot be achieved with a duplication of the number of parallel processors. Solutions to this issue are discussed in the next sub-section.

2.3.4 Increasing the activity

Increasing the activity greatly impacts the efficiency of the architecture. When the activity is maximal, the number of resources that are instantiated in order to achieve the required throughput is minimal. Its importance in the context of parallel decoding has been

demonstrated in the previous sub-section. Since the activity also depends on the schedule of the Max-Log-MAP algorithm, changing the schedule might result in higher activity. Thus, using schedule Σ^0 (or *SW*-Σ^0) can be an alternative, since its activity is less sensitive to the increase in the number of processors. But we have shown in section 2.2.5 that its activity for a single processor is actually rather low, and significantly lower than the activity of schedule *SW*-Σ^- for a frame size above 100 symbols. In the context of parallel decoding with P processors, this comparison does not lead to the same conclusion as shown in Figure 2-20. When the number of processors increases significantly, the activity of schedule *SW*-Σ^- is much lower than that of *SW*-Σ^0. For example, with N = 256, P = 8, L = 32, and l = 8, $\alpha_{SW-\Sigma^-}$ = 0.44 and $\alpha_{SW-\Sigma^0}$ = 0.6. The joint impact of the size of the frame N and the number of processors P on the two activities is studied in Figure 2-21, where the activity is given as a function of the size $M = N / P$ decoded by each processor. The inflection of the curve for the activity of schedule *SW*-Σ^- that is noticeable for P = 8 and N around 250 symbols is explained by the fact that the size of the section M becomes lower than 32. In this case, the window size L is reduced to M, which degrades the performance. The performance degradation associated with smaller window sizes will be discussed in chapter 6.

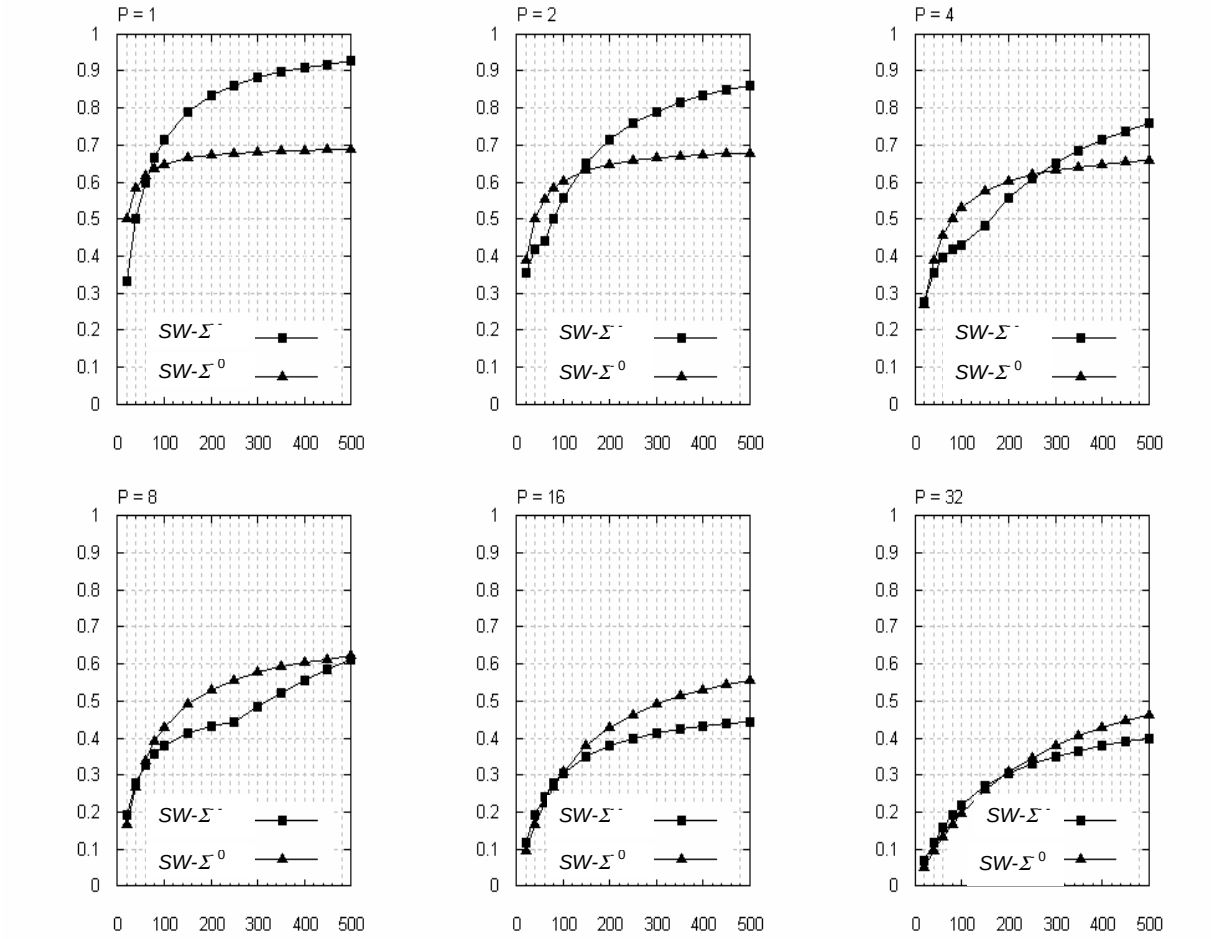


Figure 2-20: Comparison of the activities of schedules *SW*-Σ^0 and *SW*-Σ^- as a function of the frame size N and the number of processors P for L = 32 and l = 8

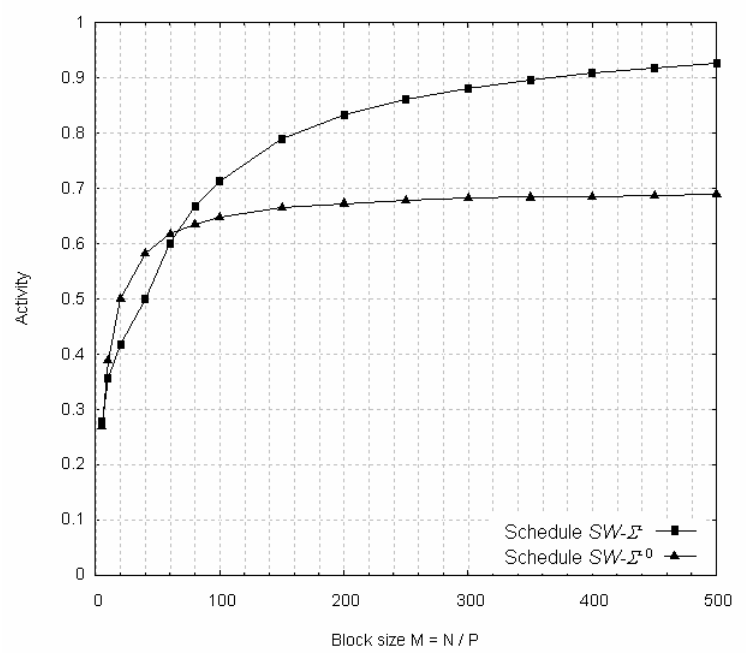


Figure 2-21: Comparison of the activities of schedules $SW\text{-}\Sigma^0$ and $SW\text{-}\Sigma^-$ as a function of the block size $M = N / P$ for $L = 32$ and $l = 8$

Even with schedule $SW\text{-}\Sigma^0$, the activity is considerably decreased when the number of processors is augmented: about half of the computational resources are not used if $P = 8$ processors decode a frame of size $N = 160$ symbols. In the context of iterative decoding of turbo codes, a significant activity improvement close to 1 could be obtained by starting the processing of the next half-iteration before the end of the current half-iteration. Nevertheless, the presence of the interleaver prevents this early processing of the next half-iteration since the extrinsic data required might not have been produced yet by the current iteration. One contribution of this thesis presented in chapter 4 is to analyse this issue and to propose solutions to overcome it.

2.3.5 Increasing the clock frequency

Improving the maximum clock frequency of turbo decoder architectures enables a proportional throughput increase. Generally, the propagation delay through the critical path that is limiting the maximum clock frequency can be reduced in two ways. The first solution relies on the utilization of faster operators, thus resulting in an increase in complexity. It is not always sufficient, though, and for some circuits such as FPGAs it is simply not feasible because of the imposed elementary structure. For these two reasons, a significant improvement in the clock frequency can be achieved by inserting more pipeline registers in the architecture. Pipelining a feed forward path can be performed easily, at the cost of an increase in latency and complexity for the additional registers. On the contrary, pipelining a feedback loop cannot be performed without changing the behaviour of the algorithm. More sophisticated techniques need to be elaborated. This problem applies for turbo decoder architectures since the critical path limiting the maximum clock frequency of the circuit

corresponds to the delay along the feedback loop in the recursions involving an Add-Compare-Select operation.

An elaborate solution, particularly suited to the decoding of double-binary code, for pipelining the feedback loop of the recursions and its impact on the behaviour of the algorithm will be presented in chapter 5.

2.3.6 Hardware design techniques

The area of a circuit can be significantly lowered with optimization of the implementation at a low level. This low-level optimization uses hardware design techniques adapted to the implementation circuit of the architecture. These techniques include displacement of operations, efficient utilization of memories, reduction of interconnections, local optimization of groups of operators, efficient use of register reset or preset signals, simplification in the control logic, look-ahead techniques, etc. Some of these techniques were applied to the implementation of a complete turbo decoder presented in chapter 6 and using the major improvements offered by the other techniques discussed previously in this section and detailed in the next chapters.

2.4 Conclusion

In this chapter, we have described the translation of the Max-Log-MAP algorithm into an efficient high-throughput hardware architecture. To this end, we have derived a relation expressing the throughput of the turbo decoder architecture as a function of the number of processors, the activity of the processors and the maximal clock frequency. Then, the influence of each term of this relation and their contribution to the throughput variations has been discussed. It has been shown that the architecture can be optimized at different levels of description.

At the algorithmic level, the choice of the algorithm and its schedule has a great impact both on the complexity and on the throughput of the architecture. Another simple method enabling a throughput increase trades off the number of iterations and the associated error correcting performance. Moreover, the association of a dynamic stopping rule with an input buffer enables a considerable increase in the throughput. This dynamic reduction of the number of iterations has been widely addressed in the literature and will not be studied in the sequel.

At the architecture level, it has been shown that the architecture efficiency can be measured by the activity of its computation units, *i.e.* a measure of the utilization of the computational resources. In fact, one of the objectives of an architecture design is to find the minimal number of computational resources required by the execution of the algorithm. It requires specification of a spatial and temporal allocation of the algorithmic operations on the instantiated hardware operators. Due to the choice of schedule of the algorithm and the inherent data dependencies, this objective cannot be achieved easily. It has been shown for the same computational complexities that architectures corresponding to two different schedules

may require different numbers of hardware operators. In fact, for a single processor implementation the activity of schedule *SW-*Σ^{-} is much higher than the activity of schedule *SW-*Σ^{0}. In this case, schedule *SW-*Σ^{-} is the most efficient.

For further enhancement of the throughput, duplication of the computational resources is mandatory. For a multiplication of the throughput by a factor of *P*, a conventional serial architecture cascades *P* component decoders, at the cost of a multiplication of the circuit complexity due to the duplication of memories for interleaving / de-interleaving. Therefore, the parallel architecture has been considered, which requires splitting the trellis associated with the decoding of one half-iteration into *P* trellis sections. Then, *P* processors decode in parallel these trellis sections. The objective that we have sought is a linear increase in throughput with the increase in computational resources. However, this objective faces two problems related to the interleaver: concurrent accesses to the memory banks and activity reduction. Concurrent accesses to the memory banks by several processor lead to the so-called parallel conflicts. Solutions to overcome this major bottleneck in parallel decoding will be described in chapter 3. Activity reduction is the other problem with parallel processing since increasing the number of processors reduces the activity and therefore the throughput of the architecture. Solutions for increasing the activity will be addressed in chapter 4.

Finally, at the gate level, an increase in the clock frequency can achieve a considerable throughput improvement. However, for turbo decoders it implies overcoming the so-called “ACS bottleneck” in the recursions computing the forward and backward state metrics (chapter 5). In addition, a reduction in the area of the circuit can be obtained by hardware optimization techniques adapted to the implementation circuit of the architecture (chapter 6).

Chapter 3

Parallel decoding of turbo codes

Contents:

Chapter 3. Parallel decoding of turbo codes

In the previous chapter, we have shown that increasing the throughput of a turbo decoder involves the utilization of several SISO processors working in parallel. However, this parallel processing leads to a bottleneck in accessing the memory banks, resulting in parallel conflicts.

In this chapter we first describe techniques that have been used to solve this problem. We will show that three types of solutions are available. They differ in the stage of the design at which they take place: either during the execution of the decoding, during the design of the architecture or during the design of the code itself. Among these three techniques, the last one is the most efficient. It involves designing jointly the interleaver and the parallel decoding architecture. This joint methodology aims at constraining the interleaver so that the resulting turbo code can be decoded in parallel without the occurrence of the above-mentioned conflicts.

The second section depicts the general parallel decoding architecture based on a two level addressing scheme, from which we will derive our constrained interleaver. The objectives in terms of complexity reduction that are associated with a constrained interleaver design will be clearly identified.

Our joint interleaver-architecture methodology is based on a new family of turbo codes called Multiple Slice Turbo Codes *(*MSTCs), which is introduced in the third section. The development of this family is a major contribution of this thesis. The interleaver corresponds to a one-to-one mapping to the two-level addressing scheme of the decoder architecture, thus yielding a two-level hierarchical interleaver (HI).

After a brief description of interleaver optimization, we analyse in the fourth section the impact of MSTC properties, and their application to the construction of various MSTCs.

Our contribution also includes the development of an optimization process of the hierarchical interleaver, which is addressed in the fifth section. The absence of performance degradation compared to conventional turbo code design is verified by simulation.

Then, the performance of several MSTCs with various frame sizes and code rates will be given in the sixth section. In particular, we will describe the MSTC that was proposed to the DVB-S2 standardization committee as a candidate error correcting code for high throughput applications.

Finally, other constructions of interleavers following the general parallel decoding architecture are discussed.

The material presented in this chapter has been published partly in [Gnaedig *et al.*-03], [Gnaedig *et al.*-05a] and has been patented [Boutillon *et al.*-02] in France.

3.1 Solutions for resolving concurrent accesses

In the previous chapter we described the problem of concurrent accesses to the memory banks that need to be solved for parallel decoding architectures. This first section aims at describing solutions to overcome this bottleneck. Basically, there are three types of solutions, characterized, by analogy with software development terminology, by their stages at which they tackle the problem of concurrent accesses (ranked from the lowest level to the highest level):

- solutions at the "execution" stage: during the execution of the parallel processing, the concurrent accesses are handled by additional hardware, which is designed to solve any occurrence of concurrent accesses (section 3.1.1).
- solutions at the "compilation" stage: during the design of the parallel decoding architecture a memory mapping preventing the emergence of any parallel conflict is found (section 3.1.2).
- solutions at the "design" stage: the interleaver and the architecture are designed jointly in order to guarantee the absence of parallel conflicts (section 3.1.3).

Even though the first two state-of-the-art solutions are flexible and can be used for any permutation law, we proposed the latter approach in our search for a low complexity hardware implementation of turbo decoders.

3.1.1 Execution-stage solutions

The first solutions that were proposed in the literature handle the parallel conflicts at the execution stage, *i.e.* during the execution of the decoding. They include additional hardware designed in such a way that all configurations of concurrent accesses can be solved. Thanks to a shuffle network, these solutions increase the memory access latency to smooth the access bandwidth to the memory banks. They exploit the third architectural law stating that communication and memorization can be exchanged (see chapter 2.1).

To illustrate this concept, let us consider the case of two write accesses to two memory banks with a latency of one cycle, depicted in Figure 3-1.a. When a collision occurs, the requirements of two write accesses per cycle cannot be satisfied. If the maximum latency is increased by one cycle, a shuffle network can be used to solve the concurrent accesses as in Figure 3-1.b. Let us consider that the shuffle network contains two consecutive sets of two write accesses. If two successive collisions occur when writing two sets of data (symbols 2 and 2', and symbols 3 and 3'), a modification of the schedule for the write accesses can solve these collisions. In fact, taking one value of each set enables the shuffle network to write the four data in the memory banks within 2 cycles. First symbols 2 and 3', and then symbols 2' and 3 are written successively to memory banks MB_0 and MB_1, respectively.

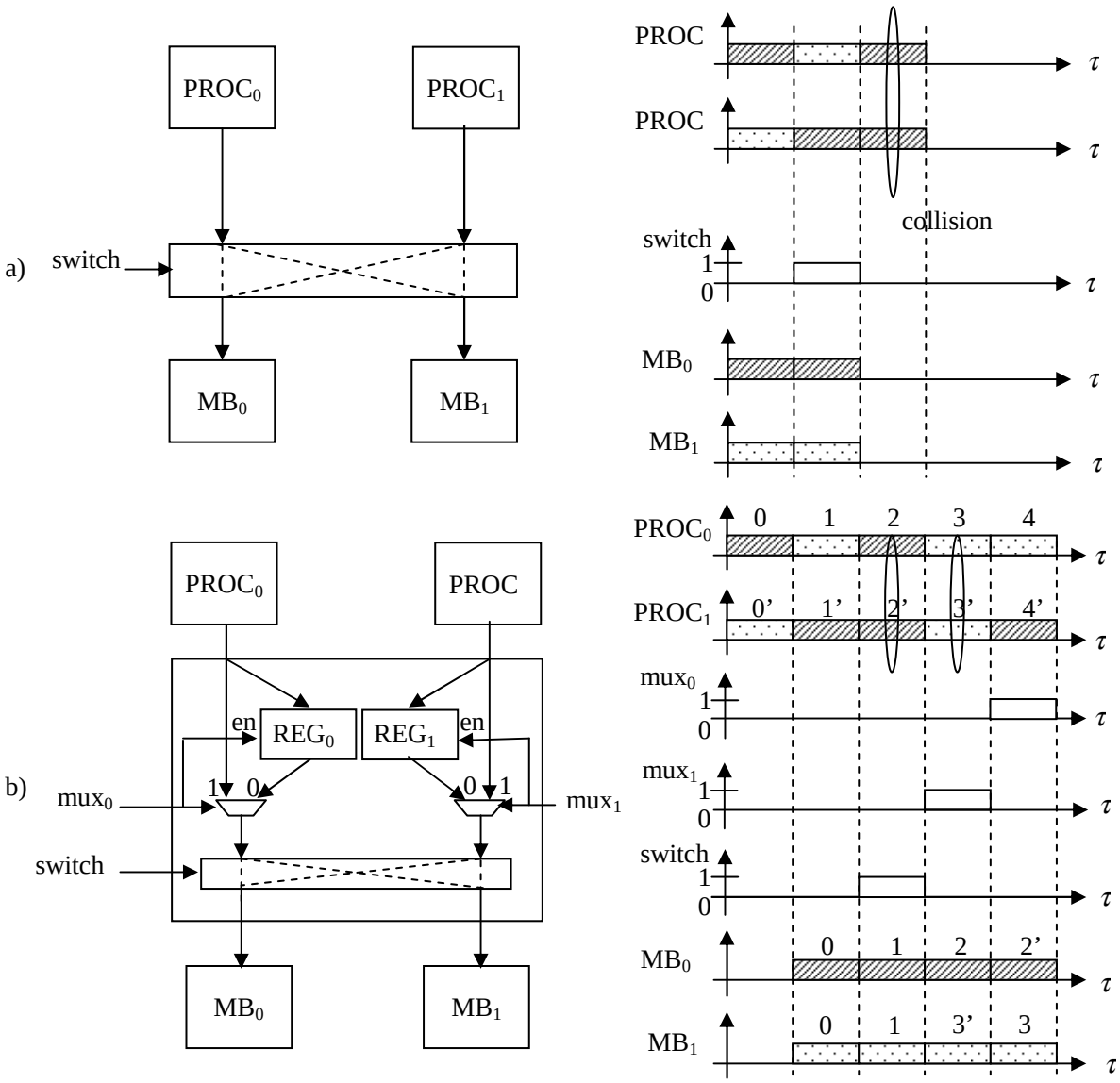
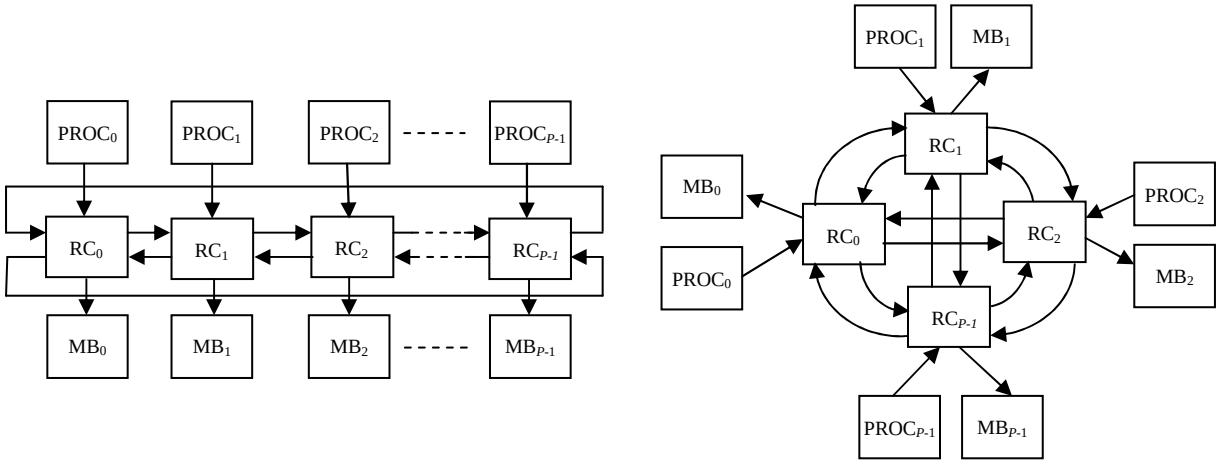


Figure 3-1: Illustration of a simple solution for resolving parallel conflicts

This simple example can be generalized by adding more latency cycles and using more complex shuffle networks. This work was done in [Thul *et al*.-02] and [Thul *et al*.-03] where several shuffle network topologies. These topologies are depicted in Figure 3-2. They are based on primitive cells called routing cells (RC), each RC being linked to a memory bank. Based on the mapping of the variable in the memory bank, an RC can either write the data into its associated memory, or transmit it to the neighbouring RCs. In [Thul *et al*.-02], the RCs are organized in a ring topology where every cell is connected to two others. It was generalized in [Thul *et al*.-03], by allowing more connections between the cells, thus resulting in a generalized network topology.



a) Ring topology b) Generalized network topology

Figure 3-2: Topologies of the shuffle network

Another construction of shuffle network was presented in [Dobkin *et al*.-02], but it allowed only a reduced set of interleavers among the $N!$ possible permutations. These techniques based on the design of a shuffle network in order to solve concurrent accesses to the same memory banks suffer from additional latency and from a non-negligible hardware penalty.

3.1.2 Compilation-stage solutions

Recently, another solution for resolving the bottleneck of the parallel decoding of turbo codes was proposed by Tarable *et al.* in [Tarable *et al*.-03] and generalized to LDPC decoding in [Tarable *et al*.-04b]. In handles this question during the design of the architecture and in particular by choosing the memory organization of the data into the memory banks. This technique performs an initial search for a mapping function φ enabling the parallel decoding in both the natural and interleaved order of the turbo code. The mapping function φ specifies for each symbol with index k in the natural order the memory bank $\varphi(k)$ in which the data (observations or extrinsic) are stored. Such a mapping function is obtained for a given schedule and satisfies the following condition: in both decoding orders, the P data accessed by the P processors are stored in distinct memory banks. Thus, all collisions in parallel decoding are avoided and no additional hardware is required. It was demonstrated that such a mapping function can be found for every interleaver. The proof is constructive and gives an algorithm that provides the mapping function guaranteeing a collision-free mapping.

Practical implementation:

An implementation of the interleaver enabling parallel accesses has been given in [Tarable *et al*.-04a]. It can be decomposed into three consecutive layers of permutations as described in Figure 3-3:

- the first layer consists of a crossbar with P inputs and P outputs. For each set of input samples produced by the processors in one dimension, it performs a "spatial" permutation according to permutation β_τ, which may be different for each time

instant τ. The attribute "spatial" refers to a shuffling of the data between distinct memory banks.

- the second layer comprises the memory banks in which the P outputs of the first layer are written using an incremental addressing scheme (0 to M-1). The transition of the data through these memory banks corresponds to an arrangement of the data according to the order that is needed for the decoding in the other dimension. To this end, P input data are read out of the memory banks with addresses given by ϕ_r, and transmitted to the third layer.
- the third layer consists of a crossbar with P inputs and P outputs. For each set of P inputs read in the memory banks, it performs a spatial permutation according to permutation β'_τ. The P outputs are then transferred to the P processors.

It has been shown in [Tarable *et al*.-04a] that the functions β_τ, ϕ_r and β'_τ can be obtained easily from the memory mapping function φ. A similar decomposition of the interleaver into three layers has been used recently in [Prescher *et al*.-05] to build a parallel turbo decoder on an ASIC circuit. This decomposition of an interleaver into three routing layers had already been addressed in [Leighton-92] in the more general area of parallel architectures. But this work has remained unknown until now to the turbo decoder design community.

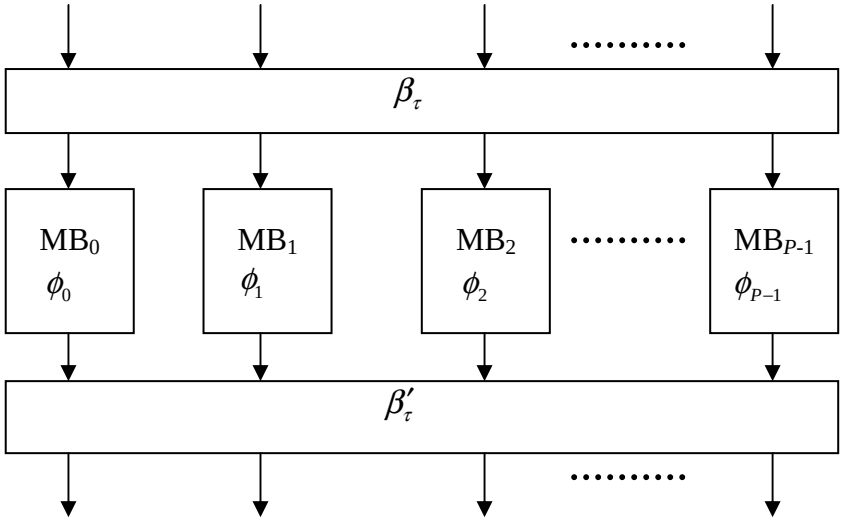


Figure 3-3: Decomposition of the parallel interleaver into three successive permutations

A straightforward implementation of the crossbar instantiates P multiplexers $P \rightarrow 1$, one for each output. This results in a total of P (P – 1) multiplexers $2 \rightarrow 1$. However, since the crossbar does not route several inputs to the same output, the number of multiplexers is in excess and other routing topologies can be used. In [Prescher *et al*.-05], it was proposed to use the Beneš network. A Beneš network permutes P input data in any order as long as two data are not destined for the same output. The network connects P inputs to P outputs using $2(\log_2 P)$-1 stages with P/2 switches at each stage. Since each switch requires 2 multiplexers $2 \rightarrow 1$, the total number of multiplexers is $P(2(\log_2 P)-1)$.

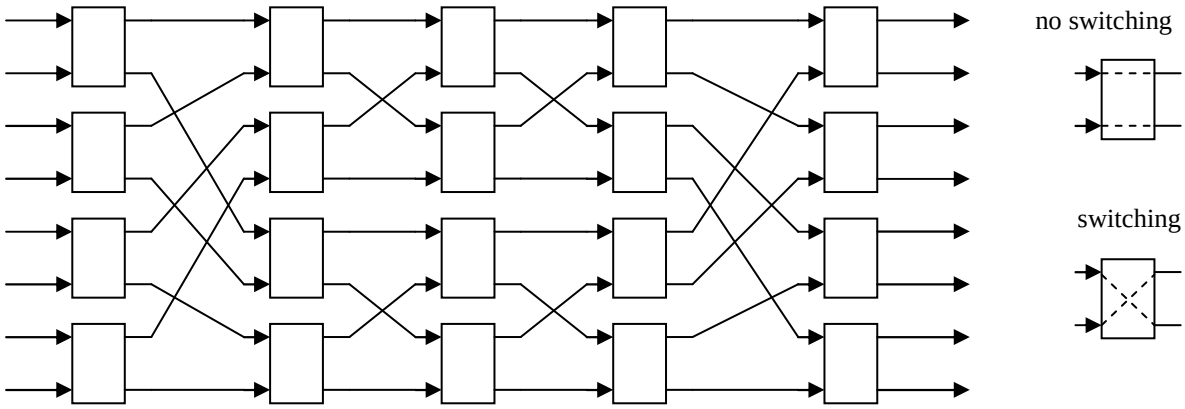


Figure 3-4: Beneš network for $P = 8$

Drawbacks:

The compilation-stage solution involves storing the memory mapping, which can represent an important amount of memory for large block sizes. This memory requirement is even more critical in the case of a scalable architecture that can decode several frame sizes resulting in as many different interleavers. It is claimed in [Tarable *et al*.-04b] that the complexity of this algorithm is low and that the memory mapping can be generated on the fly in a pre-processing step. But, at least one memory is needed to store the current memory mapping for the two dimensions of the code. In addition, for interleavers that can be described with closed-form equations, this solution spoils the reduced memory requirements for storing the interleaver parameters.

3.1.3 Conception-stage solutions

The two techniques handling the bottleneck of parallel decoding described above – shuffle network design and search for memory mapping – explore the architecture possibilities at the decoder side. They are well suited for designing parallel architecture when the interleaver is fixed. On the other hand, when the encoding scheme is not fixed, like for proprietary applications or during the elaboration process of a new standard, an additional degree of freedom in the choice of the interleaver is left to the designer. In this case, a more efficient parallel decoding architecture can be obtained by a joint code-architecture design methodology. This joint methodology relies on the design of the interleaver such that a parallel architecture can be used without parallel conflicts. This objective can be achieved with structured interleavers, where the interleaver structure enables us to find an appropriate memory mapping allowing parallel decoding. The difference with the solution proposed by Tarable *et al.* is that a trivial memory mapping is known during the interleaver design step and no algorithm is used to find it. It will be shown in the next section that this makes it possible to reduce the complexity of the parallel interleaver implementation.

We proposed a joint interleaver architecture design methodology in [Boutillon et *al*.-02] and [Gnaedig *et al*.-03] together with a modification in the constituent codes. This new family of turbo codes suitable for parallel decoding was called Multiple Slice Turbo Codes. Other joint interleaver-architecture design methodologies have also been proposed independently in [Giulietti *et al*.-02a], [Nimbalker *et al*.-03] and [Kwak *et al*.-03] using conventional RSC

binary component encoders. Since our methodology was new and enabled us to solve an important problem in parallel turbo decoding, this family of turbo codes was patented in France [Boutillon *et al*-02].

Let us start with a general description of the parallel architecture that will be used for our constrained interleaver design.

3.2 Parallel decoding architecture

In this section the characteristics of an efficient architecture suitable for parallel decoding of turbo codes is described. It will be used for our constrained interleaver design presented in 1.3 and also for other parallel interleavers discussed in 1.7. A similar model has already been used for the design of efficient LDPC decoding architectures in [Boutillon *et al.*-00]. We extended it to the design of efficient turbo decoder architectures in [Boutillon et *al*.-02] and [Gnaedig *et al.*-03]. The joint interleaver-architecture design in [Giulietti *et al.*-02a], and more recently in [Nimbalker *et al*.-03] and [Kwak *et al*.-03] have proposed the same model independently.

3.2.1 Architecture description

In chapter 2.3, we described an architecture where P processors perform the same schedule. Each memory (observation or extrinsic memory) is tiled into a set of P memory banks to enable P concurrent accesses. From this parallel architecture, constraints imposed on the interleaver design will be derived to prevent concurrent accesses to the same memory bank.

A simple architecture for parallel decoding using P processors is depicted in Figure 3-5. For the sake of a clearer exposition, this figure represents only a single datapath between a single set of memory banks and the P processors. This datapath allows P simultaneous accesses to the P memory banks. Each set of memory banks (observation or extrinsic) is associated with one or more similar datapaths. In addition, if the schedule requires m_a accesses per cycle ($m_a \leq 2$) to each set of memory banks, m_a identical datapaths are used, thus enabling $P \cdot m_a$ simultaneous accesses. For the extrinsic memory, the datapath can be dedicated either to read accesses or write accesses. Consequently, if the schedule of the algorithm performs read and write operations to a single port, the datapath needs to be duplicated. In the figure, the datapath is described for read accesses only.

This architecture comprises P processors $PROC_0$, $PROC_1$,…, $PROC_{P-1}$ working in parallel. Each processor is dedicated to the decoding of one block of size M in each dimension of the code. The architecture further comprises P memory banks MB_0, MB_1,…, MB_{P-1} addressed by their respective Memory Address Generators MAG_0, MAG_1,…, MAG_{P-1} providing at time τ the addresses to read the values stored in the memory banks. The data set read from the memory and transferred to the processors is permuted by means of a crossbar called a Spatial Permuter (SP). This permuter shuffles a single set of P data and does not mix data of two or more consecutive sets, contrary to the shuffle networks described in section 3.1.1.

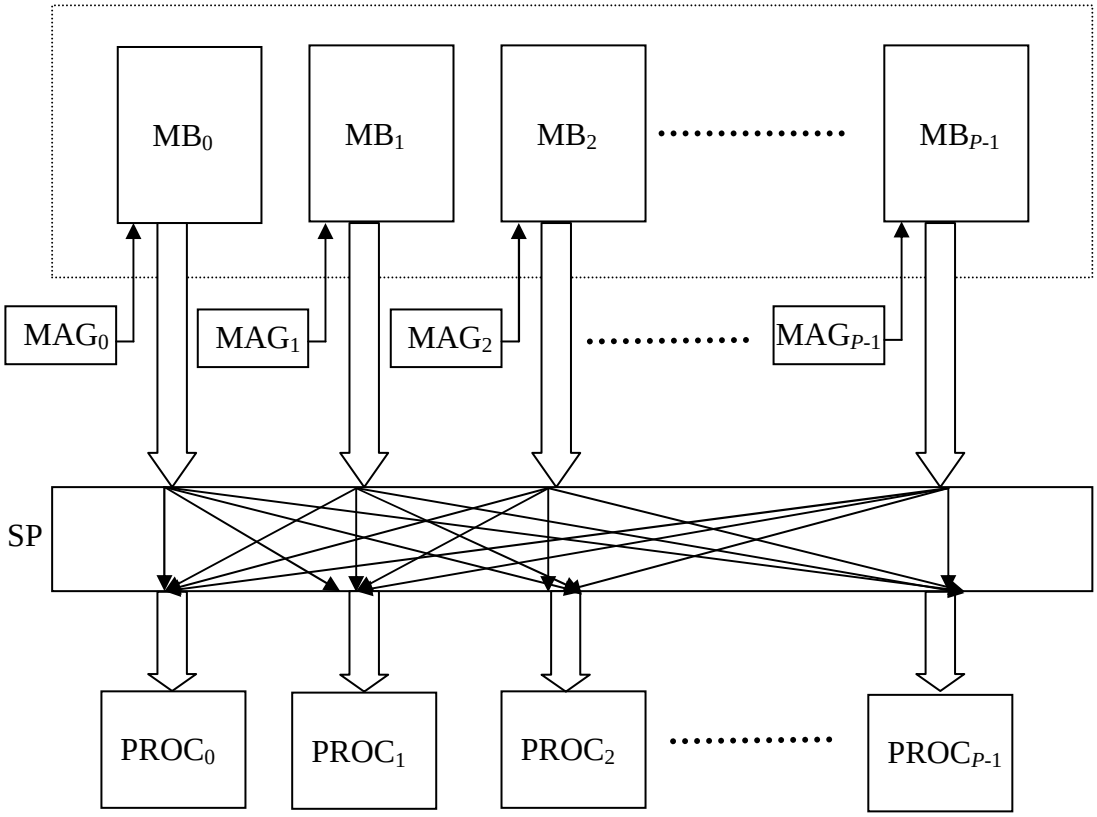


Figure 3-5: Datapath for a parallel architecture for turbo decoding

3.2.2 Objective of constrained interleaver design

We have seen in section 3.1.2 that the work of Tarable *et al.* makes it possible to find a memory mapping whatever the choice of interleaver, such that the resulting turbo code can be decoded on the parallel architecture described in Figure 3-5. In this case, the Spatial Permuter corresponds to a Beneš network allowing any permutation of P data. The cost that is paid for this parallel interleaving concerns the complexity of this network and the memory required to store the indices of the permutations (Memory Address Generators and Spatial Permuter).

Our aim is to find an interleaver Π associated with an evident and structured memory mapping such that no memories are needed to store it. In particular, when the interleaver is expressed with closed-form equations, simple dedicated circuitry will be used to control the parallel interleaving scheme, *i.e.* the generation of the Memory Address Generators and the Spatial Permuter (SP), expressed as closed-form relations. In addition, the objective of our constrained-interleaver design is to decrease the complexity related to the implementation of the SP. To this end, only a reduced set of permutations among the $P!$ possible permutations are allowed. Although this approach certainly reduces the set of interleavers that can be decoded on such a parallel architecture, it will be shown in the next sections that this constraint does not degrade the performance of the turbo code. The reduction in the SP complexity is relevant since it is duplicated several times, once for each datapath.

Let us now describe the family of turbo codes called Multiple Slice Turbo Codes that can be decoded on such a parallel architecture.

3.3 Multiple Slice Turbo Codes

Multiple Slice Turbo Codes (MSTCs) were introduced in [Gnaedig *et al.*-03] and described in detail in [Gnaedig *et al.*-05a]. MSTCs rely on two main characteristics used for the constrained interleaver design:

- a modification of the constituent convolutional codes exhibiting more parallelism and thus adapted to parallel decoding.
- a “hierarchical” interleaver design associated with the hardware datapath for accessing the memory banks in the interleaver order.

3.3.1 Constituent codes and parallel decoding

Multiple Slice Turbo Codes are constructed as described in Figure 3-6. An information frame of N m-binary symbols is divided into K blocks (called "slices") of M symbols, where $N = M \cdot K$. The resulting turbo code is denoted (N, M, K). As with a conventional convolutional two-dimensional turbo code, the coding process is first performed in the natural order to produce the coded symbols of the first dimension. Each slice is encoded independently with a CRSC code. The information frame is then permuted by an N symbol interleaver. The permuted frame is again divided into K slices of size M and each of them is encoded independently with a CRSC code to produce the coded symbols of the second dimension. Puncturing is applied to generate the desired code rate.

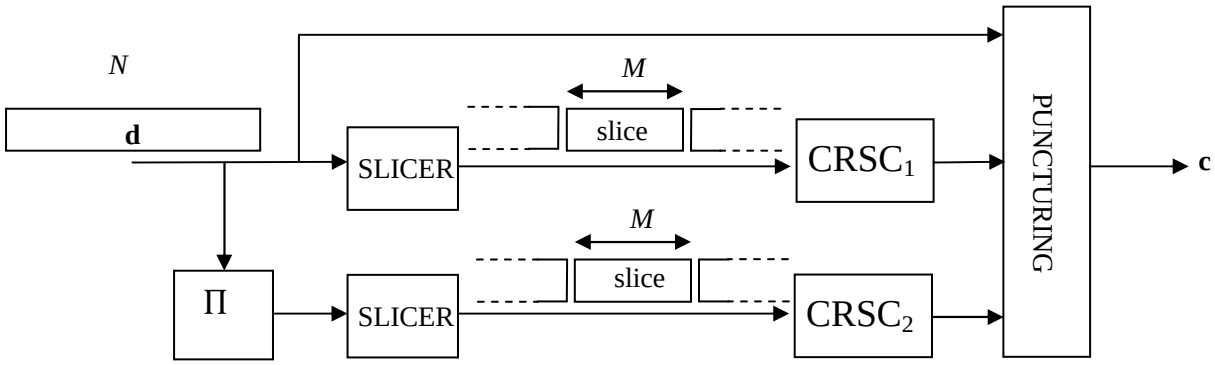


Figure 3-6: Encoding scheme of two dimensional Multiple Slice Turbo Codes

The choice of CRSC codes for encoding the slices is obvious. In chapter 1.5, several trellis termination techniques were considered. Constructing circular trellises represents the most efficient solution in terms of spectral efficiency and performance. Since several trellises are used, the interest of this technique is even greater. Therefore, using CRSC codes results in a uniform protection of all symbols of the frame, without loss in transmission efficiency. A similar approach of slicing the constituent codes into several independent trellises has also been proposed recently in [Wang *et al.*-03]. For each trellis, the encoder starts from the all-zero state and is driven into the all-zero state using tail symbols. These symbols are not interleaved and therefore this solution is penalized by the unequal error protection of the

symbols in the frame. In addition, there is a loss of efficiency in transmission when the number of slices is large.

The value of K can be chosen according to the requirements of the application. In other words, K is chosen in order to correspond to the number of parallel processors P satisfying the required throughput. However, the number of slices K and their size also influence the performance of the turbo code and can thus be chosen independently. This concern will be addressed in section 3.4.3. Unless the contrary is explicitly stated, we consider the case where the number of slices equals the number of processors, $K = P$, and we use the unique notation P to designate the number of slices and the number of processors.

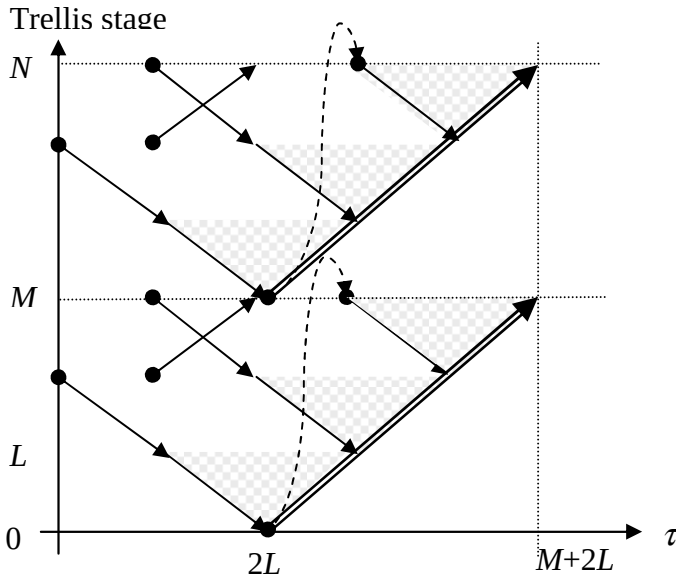


Figure 3-7: Parallel decoding of MSTC by 2 processors using schedule *SW-Σ⁻* and pre-processing steps

Cutting the constituent code into slices was first devised for exhibiting distinct trellis sections. Indeed, there is no direct relation between two symbols of distinct slices of one dimension. This property of independent trellises can be used in order to decode the P slices simultaneously by P processors. Figure 3-7 describes the parallel decoding of the slices using schedule *SW-Σ⁻*. Since each trellis is independent, the recursions for each slice need to be initialized. This is done in Figure 3-7 with a pre-processing step computing the circular state for each of the K trellises (see chapter 1.5). In Figure 3-8, the low complexity NII technique (see chapter 1.5) is used to initialize the recursions. This shows that the schedule for decoding a sliced turbo code does not differ significantly from the parallel schedule *SW-Σ⁻* with P processors described in chapter 2.3 for a single-slice turbo code (SSTC). The only difference lies in the initialization of the state metrics at the boundaries of the trellis section: with multiple circular trellises, the state metrics are exchanged within each slice, whereas with a single trellis, they are exchanged with adjacent trellis sections of the same trellis. Consequently, for identical decoding schedules, the error decoding performance of the MSTC designed with P slices and decoded by P decoders is equivalent to the performance of the SSTC decoded in parallel by P processors.

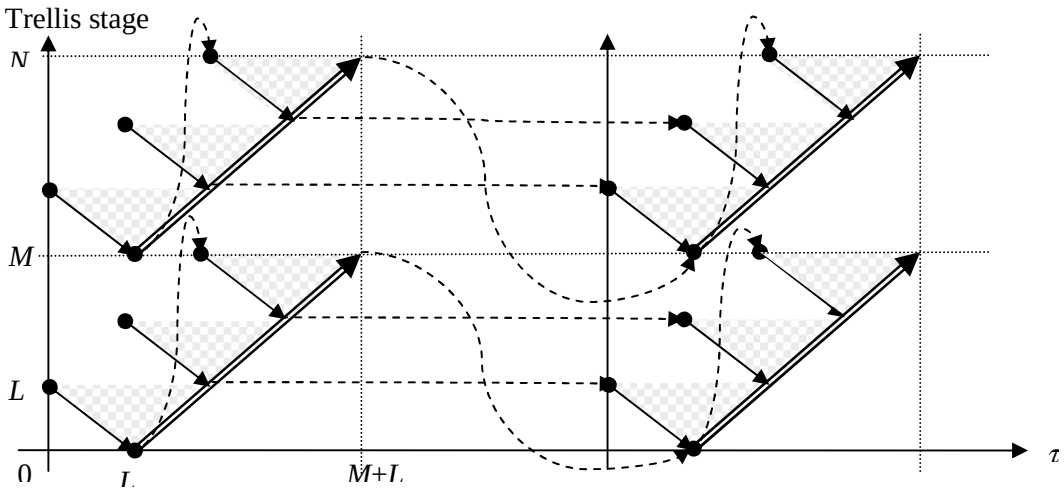


Figure 3-8: Parallel decoding of MSTC by 2 processors using schedule *SW-Σ*$^{-}$ and the NII technique

Note also that MSTCs have been used successfully for the implementation of a turbo decoder on a DSP with Single Instruction Multiple Data (SIMD) capability [Gnaedig *et al.*-04b] (see also Appendix E). In addition, the slicing property of MSTCs are of great interest for the analog decoding of turbo codes and first promising results are shown in [Arzel *et al.*-05]. The slicing of the constituted codes can also be used to design easily a "hierarchical" interleaver suited to the decoding architecture presented in the previous section. Such an interleaver ensuring parallelism constraints while maintaining good performance is described in the next section.

3.3.2 Interleaver construction

In this section, we present the construction of the interleaver associated with MSTCs that guarantees the absence of parallel conflicts.

Let Π denote the interleaving function that links each symbol with index k of the natural order to the symbol with index $\Pi(k)$ in the interleaved order. As already discussed in chapter 2.3, the interleaver construction is associated with a memory mapping of symbols into the memory banks. For MSTCs, a straightforward mapping function is used: the symbols of slice r in the natural order are all mapped to the memory bank MB_r. Then, the construction of the interleaver for MSTCs follows exactly the structure of the parallel decoding architecture presented in Figure 3-5, using an interleaver separated into two hierarchical levels:

- the first level corresponds to the Spatial Permuter and exchanges the data between the memory banks. It is defined by a permutation Π_S called a spatial permutation, which therefore exchanges data between slices.
- the second level corresponds to the Memory Address Generators MAG_0, MAG_1,…, MAG_{P-1} and shuffles the data within each memory bank. This second level is defined, respectively, by P permutations Π_T^0, Π_T^1,…, Π_T^{P-1} called temporal permutations shuffling the symbols within each slice.

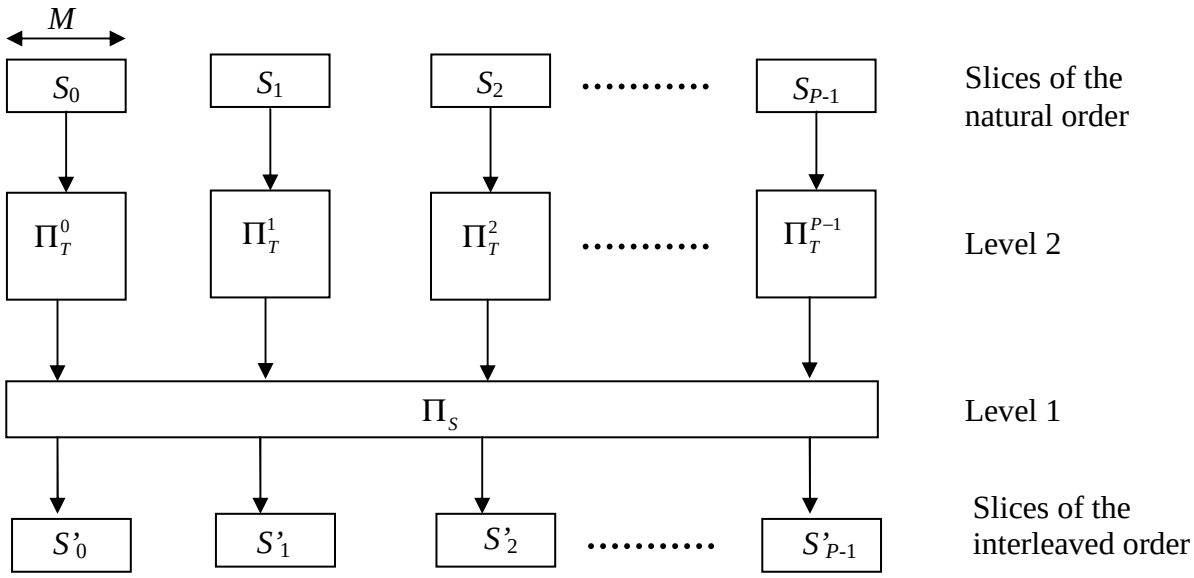


Figure 3-9: A two level hierarchical interleaver for MSTCs

Hierarchical interleaver:

The separation of the resulting interleaver Π into two levels of permutations explains the hierarchical property of the interleaver. The corresponding hierarchical interleaver (HI) is denoted HI(*P*).

Let *l* and *k* denote the indices of the symbols in the natural and interleaved order, respectively. The coding process is performed in the natural order on independent consecutive blocks of *M* symbols. The symbol with index *l* (*l* = 0,…, *N*-1) belongs to slice $\lfloor l / M \rfloor$, and is the (*l* mod *M*)th symbol of the slice. Likewise, in the interleaved order, the symbol with index *k* (*k* = 0,…, *N*-1) belongs to slice $r = \lfloor k / M \rfloor$ and is the $(t = k \bmod M)^{th}$ symbol of the slice. Note that $k = M \cdot r + t$, where $r \in \{0,\ldots,P-1\}$ and $t \in \{0,\ldots,M-1\}$. For each symbol with index *k* in the interleaved order, the permutation Π associates a corresponding symbol in the natural order with index $l = \Pi(k) = \Pi(t,r)$. The hierarchical design splits the interleaver function into two levels: the spatial permutation $\Pi_S(t,r)$ (ranging from 0 to *P*-1) and the temporal permutation $\Pi_T(t,r)$ (ranging from 0 to *M*-1), as defined in (3-1) and described in Figure 3-9.

$$\Pi(k) = \Pi(t,r) = \Pi_S(t,r) \cdot M + \Pi_T(t,r) \qquad \textbf{(3-1)}$$

The symbol at index *k* in the interleaved order corresponds to the symbol in slice $\Pi_S(t,r)$ at address $\Pi_T(t,r)$. While decoding the first dimension of the code, the frame is processed in the natural order. The spatial permuter and Memory Address Generators are then simply replaced by *Identity* functions.

Let ρ be the division of the number of symbols in a slice by the number *P* of slices: $\rho = M / P$. This parameter will be used later on to characterize MSTCs. In what follows we assume, without loss of generality, that ρ is an integer. This implies that two distinct slices

with index j in the first dimension and index i in the second dimension share exactly ρ symbols with temporal indices $\tau_{i,j} = \{\mathbf{T}(i,j) + q \cdot P\}_{q=0,\ldots,\rho-1}$ in the interleaved order, where $\mathbf{T}(i,j) \in \{0,\ldots,P-1\}$ is the temporal index that verifies $\Pi_S(\mathbf{T}(i,j),i) = j$, and $\mathbf{T}$ is a square matrix of size P.

Temporal permutation:

In order to simplify the hardware implementation, the same temporal permutation is chosen for all slices. Thus, $\Pi_T(t,r) = \Pi_T(t)$ depends only on the temporal index t. This solution has the advantage of requiring one single address computation to read P data symbols from the P memory banks. It also simplifies the complexity of the interleaver design task by reducing the number of parameters in the interleaver.

Spatial permutation:

The spatial permutation shuffles P data extracted from the slices in the natural order and transferred to the slices of the interleaved order. Decoder r receives the data from slice $\Pi_S(t,r)$ at instant t. For each fixed t, function $\Pi_S(t,r)$ is then a bijection from the decoder index $r \in \{0,\ldots,P-1\}$ to the slices $\{0,\ldots,P-1\}$.

Furthermore, to maximize the shuffling between the natural and the interleaved order, we constrain function $\Pi_S(t,r)$ such that every P consecutive symbols of a slice of the interleaved order comes from P distinct slices of the natural order. To this end, for a given r, function $\Pi_S(t,r)$ is chosen to be bijective and P-periodic. The bijectivity means that $\Pi_S(t,r)$ is a bijection from the temporal index $t \in \{0,\ldots,P\text{-}1\}$ to the set $\{0,\ldots,P\text{-}1\}$ of slice indices. The P-periodicity of the temporal index means that for $\forall t$, $\forall q$ satisfying $t + q \cdot P < M$, one obtains $\Pi_S(t + q \cdot P, r) = \Pi_S(t,r)$. In the rest of the manuscript only bijective P-periodic functions are considered for the spatial permutation. In particular, we restrict ourselves to a circular shift of amplitude A, whose implementation has a low complexity. When P is a power of two, it is expressed as

$$A = \sum_{h=0}^{\log_2(P)-1} a_h 2^h \,. \qquad \textbf{(3-2)}$$

Its implementation depicted in Figure 3-10 leads to $\log_2(P)$ levels of multiplexers, where each level h, $h = 0,\ldots, \log_2(P)\text{-}1$ performs a circular shift of amplitude 2^h when $a_h = 1$. No shift is performed otherwise. Since each level is composed of P multiplexers $2 \rightarrow 1$, the complexity of the circular shift is $P \cdot \log_2(P)$ multiplexers, which is around half the complexity of the Beneš network. In addition, the circular shift is controlled by $\log_2(P)$ bits, compared to $\frac{P}{2}(2\log_2(P)-1)$ for the Beneš network.

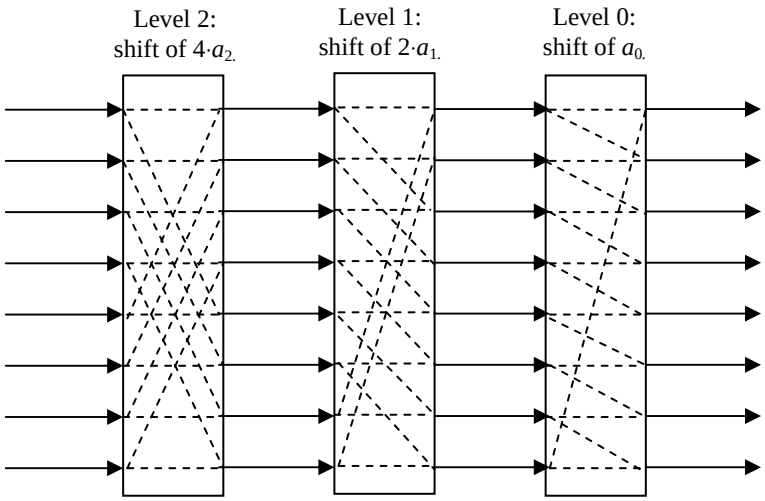


Figure 3-10: Implementation of a circular shift (*P*=8)

3.3.3 Example

Let us construct a simple (*N*, *M*, *P*) = (18,6,3) code to clarify the HI construction. Let the temporal permutation be $\Pi_T(t) = \{1,4,3,2,5,0\}$ (*i.e.* $\Pi_T(0) = 1$, $\Pi_T(1) = 4$, …) and let the spatial permutation be a circular shift of amplitude $A(t \bmod 3)$, *i.e.* the slice of index *r* is associated with the slice of index $\Pi_S(t,r) = (A(t \bmod 3) + r) \bmod 3$, with $A(t \bmod 3) = \{0,2,1\}$. The spatial permutation is then bijective and 3-periodic.

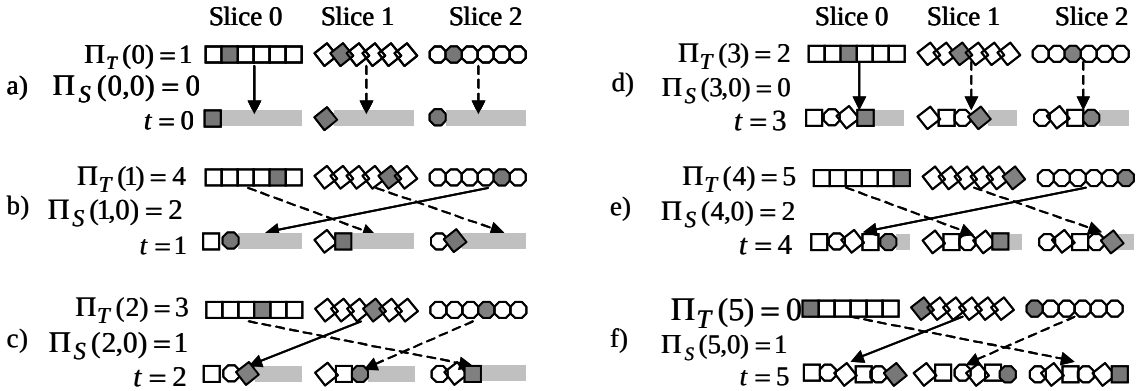


Figure 3-11 A basic example of a (18, 6, 3) code with $\Pi_T(t)$ = {1,4,3,2,5,0} and $A(t \bmod 3)$ = {0,2,1}

The interleaver is illustrated in Figure 3-11, which shows the permutations for the 6 temporal indices *t* = 0 (a), *t* = 1 (b), *t* = 2 (c), *t* = 3 (d), *t* = 4 (e) and *t* = 5 (f). The 18 symbols in the natural order are separated into 3 slices of 6 symbols corresponding to the first dimension. In the second dimension, at temporal index *t*, symbols $\Pi_T(t)$ are selected from the 3 slices of the first dimension, and then permuted by the spatial permutation $\Pi_S(t,r)$. For example, at temporal index $t = 1$ (b), symbols at index $\Pi_T(1) = 4$ are selected. Then, they are shifted to the left with an amplitude $A(1 \bmod 3) = 2$. Thus, symbols 4 from slices 0, 1 and 2 of the first dimension go to slices 1, 2 and 0 of the second dimension, respectively.

3.4 Design of Multiple Slice Turbo Codes

After a brief overview of interleaver optimization, we analyze the influence of slicing on the interleaver design and on the performance of the turbo code. Then, we will present two code designs using MSTCs.

3.4.1 Overview of interleaver optimization

As described in chapter 1.3, the design of the interleaver aims to fulfil two performance criteria. The first is to achieve the convergence of the code at low SNRs, which is influenced by the presence of cycles in the interleaver. The second is to obtain a good minimum distance for the asymptotic performance of the code at high SNRs. In fact, a high minimum distance is needed to lower the "error floor" induced by the presence of low-weight codewords. After an analysis of the cycles in an interleaver and their characterizations, their impact on low-weight codewords will be analyzed through a study of the low-weight sequences called Return To Zero sequences. Then the impact on the interleaver design of slicing the component codes into multiple independent trellises will be discussed.

3.4.1.1 Cycles in an interleaver and their characterization

This section defines the cycles of an interleaver and characterizes their impact on the performance of the turbo code.

Definition 3-1: The norm $|x|_M$ is defined as $|x|_M = \min(x \bmod M, -x \bmod M)$.

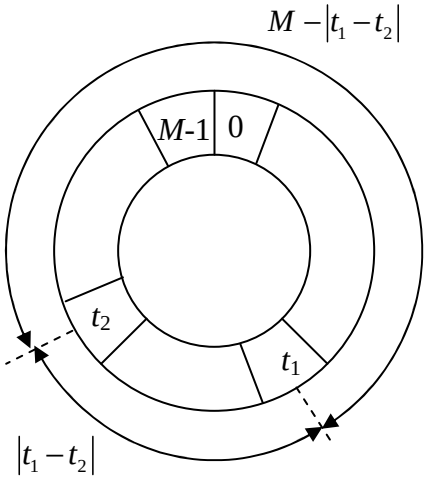


Figure 3-12: Distance between two symbols with indices t_1 and t_2 in a slice

Definition 3-2: The distance $d_M(t_1,t_2)$ between two symbols k_1 and k_2 of a same slice, and whose temporal indices are denoted t_1 and t_2 respectively, is defined by the length of the shortest path $|t_1-t_2|_M = \min(|t_1-t_2|, M-|t_1-t_2|)$ between the two symbols (see Figure 3-12).

In the general case, two symbols located in two distinct slices do not belong to the same trellis and do not impact on each other in this dimension: their distance will be assumed to be infinity. Thus, the definition of distance is generalized by:

Definition 3-3: The distance $\|k_1 - k_2\|_M$ between two symbols k_1 and k_2 of the frame is defined by:

$\|k_1 - k_2\|_M = |k_1 - k_2|_M$ if symbols k_1 and k_2 are in the same slice (see Figure 3-13.b).

$\|k_1 - k_2\|_M$ = infinity otherwise (see Figure 3-13.a).

In the following, for the sake of simplicity, we shall not specify whether we refer to definition 2 or definition 3. The distance $\| \|_M$ is used for the distance between two symbols, whereas $| |_M$ is used for the distance between the temporal indices of the symbols in a slice. Thus, $\|k_1 - k_2\|_M = |t_1 - t_2|_M$ if symbols k_1 and k_2 are both in the same slice with temporal indices t_1 and t_2.

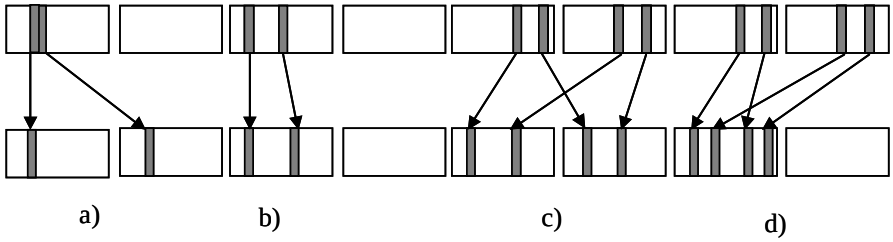


Figure 3-13: Primary and secondary cycles:
a) no cycle, b) primary cycle, c) external secondary cycle, d) internal secondary cycle.

Definition 3-4: A primary cycle is defined as a set of two symbols that are in a same slice in both the natural and interleaved order, as shown in Figure 3-13.b.

It is natural to analyze the impact on the performance of a primary cycle by its spread defined as the sum of the distance between the symbols in the two dimensions [Crozier *et al.*-00]:

$$S(k_1, k_2) = \|k_1 - k_2\|_M + \|\Pi(k_1) - \Pi(k_2)\|_M \quad \textbf{(3-3)}$$

The spread $S(k_1, k_2)$ between two symbols that belong to a primary cycle is related to the correlation between the extrinsic information inputs associated with symbols k_1 and k_2. The higher the spread, the lower the correlation. Thus, the minimum spread S among all possible primary cycles, $S = \min_{k_1, k_2} [S(k_1, k_2)]$, should be reasonably high in order to avoid both short primary cycles and thus high correlation. We have demonstrated in [Boutillon *et al.*-05] that for a conventional (with no slices) two-dimensional turbo code the spread is bounded by $\sqrt{2N}$. The geometrical demonstration using a volume argument is reproduced in Appendix B. We have also demonstrated that for MSTCs with $\rho \geq 2$ (*i.e.* $P \leq M$) the maximal spread remains unchanged.

Definition 3-5: A secondary cycle is defined as a set of 4 symbols that constitute two couples in the two dimensions of the code as shown in Figure 3-13.c, d.

Definition 3-6: A secondary cycle is said to be internal when the four symbols are in the same slice in at least one of the two dimensions (Figure 3-13.d). It is said to be external in the other case (Figure 3-13.c).

To characterize secondary cycles, we extend the notion of spread using the summary distance [Thruhachev *et al*.-01]. For each secondary cycle, there are four couples of symbols that belong to four slices or less: slices i_1 and i_2 in the interleaved order, slices j_1 and j_2 in the natural order. Their temporal indices are denoted $\left(u_{i_1}^1, u_{i_1}^2\right)$ and $\left(u_{i_2}^1, u_{i_2}^2\right)$ in the slices of the interleaved order, and $\left(v_{j_1}^1, v_{j_1}^2\right), \left(v_{j_2}^1, v_{j_2}^2\right)$ in the slices of the natural order. The summary distance of a secondary cycle is then defined as the sum of the 4 pair-wise lengths between the temporal indices of the 4 couples:

$$d_{sum} = \left|u_{i_1}^1 - u_{i_1}^2\right|_M + \left|u_{i_2}^1 - u_{i_2}^2\right|_M + \left|v_{j_1}^1 - v_{j_1}^2\right|_M + \left|v_{j_2}^1 - v_{j_2}^2\right|_M \qquad \textbf{(3-4)}$$

Definition 3-7: A secondary cycle is said to be "rectangular" if $\left|u_{i_1}^1 - u_{i_1}^2\right|_M = \left|u_{i_2}^1 - u_{i_2}^2\right|_M$ and $\left|v_{j_1}^1 - v_{j_1}^2\right|_M = \left|v_{j_2}^1 - v_{j_2}^2\right|_M$.

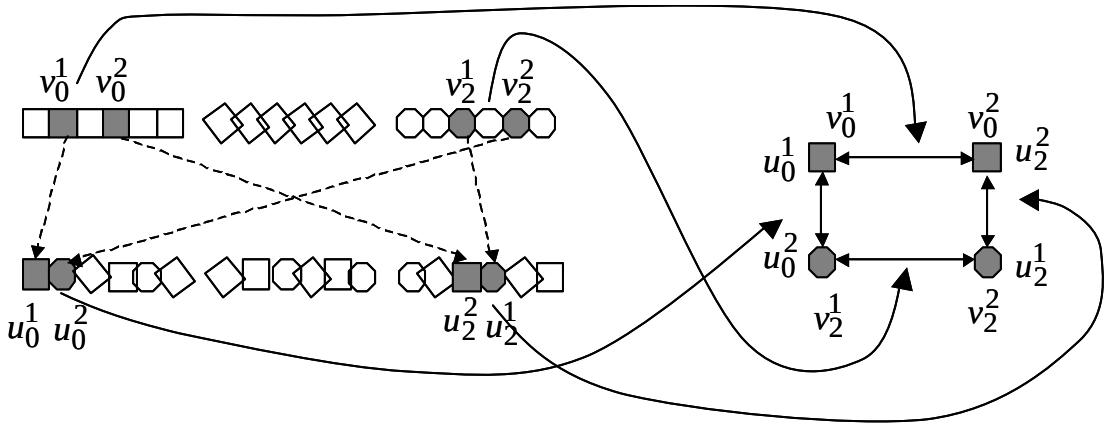


Figure 3-14: External rectangular secondary cycle representation in the example of section 3.3.3 with $(i_1, j_1, i_2, j_2) = (0,0,2,2)$

For a rectangular secondary cycle, the distances between the symbols of the two couples of the first dimension are equal, as presented in Figure 3-14. It is the same for the couples of the second dimension.

The optimization of the interleaver aims to find appropriate temporal and spatial permutations. The design of these permutations is performed by analyzing primary cycles characterized by their spread, and secondary cycles and patterns characterized by their summary distance. The impacts on these cycles on low-weight codewords are analyzed in the next section.

3.4.1.2 RTZ sequences and low-weight codewords

Definition 3-8: For an *m*-binary convolutional code, a Return To Zero (RTZ) sequence is defined as an ordered finite list of symbols that makes the encoder diverge from state 0 and then return to state 0. The number of symbols after the first symbol in the list defines the length of the sequence.

An RTZ input sequence produces a finite number of parity bits. The weight of an RTZ sequence is then equal to the number of 1s in the input bits and in the corresponding parity bits. These RTZ sequences represent the low-weight error sequences of a convolutional code. As a first approximation, we will consider that their weight grows linearly with their length. For an 8-state recursive systematic binary convolutional code (rate 1/2) with memory $\nu = 3$ and with generator polynomials 15 (feedback) and 13 (redundancy), the RTZ sequence of shortest length with input weight 2 is the sequence of length 7: 10000001. The weight of its parity bits is equal to 6. For the 8-state double-binary RSC code described in chapter 1.2, it was shown in [Berrou *et al*.-99b] that there are other basic RTZ sequences of different lengths. The input couple of the convolutional encoder can take 4 different values denoted **0** = (0,0), **1** = (0,1), **2** = (1,0), **3** = (1,1). For this code, the primitive RTZ sequences are 13, 201 and 30002. Without puncturing, the weight (information and parity bits) of these sequences are 4, 5 and 7, respectively (see [Berrou *et al*.-99b] for more details about double-binary turbo codes).

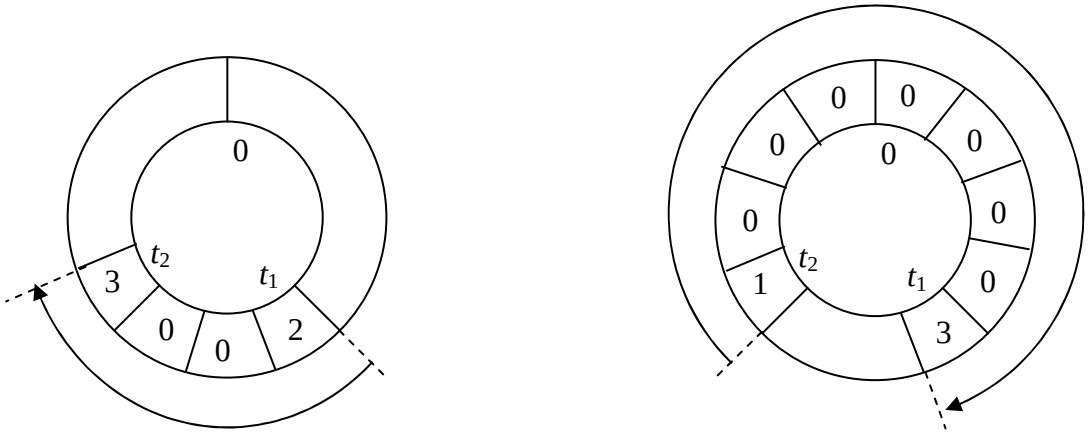


Figure 3-15: Example of RTZ sequences for two given symbols with the double-binary CRSC code with a slice of size $M = 11$

Because of the tail-biting technique, and especially for short slices, two different RTZ sequences can be generated by a given pair of non-zero symbols in a slice (see Figure 3-15). In a first approximation, the RTZ sequence with the shortest length will be considered. In fact, this RTZ sequence may have the smallest Hamming weight and thus, penalizes the minimum distance the most. Hence, for two symbols in a slice, the RTZ sequence of smallest weight (if there are any) is proportional to the distance (definition 3) between the two symbols.

If the two symbols of a primary cycle correspond to an RTZ sequence in the two dimensions, then the primary cycle is also called a Primary Error Pattern (PEP). The minimum weight of a potential PEP is approximately proportional to the spread between two symbols that belong to a primary cycle. Thus PEPs are strongly related to primary cycles: short primary cycles may lead to low-weight codewords, while long primary cycles lead to higher weight codewords. Thus, maximizing the spread minimum spread S avoids low-weight PEPs. Like for the primary cycle, we can define a secondary error pattern (SEP) as a secondary cycle whose symbols form two RTZ sequences in the two code dimensions. Consequently, maximizing the summary distance of the secondary cycles leads to increasing the weight of SEPs.

Therefore optimizing the spread of the interleaver and the summary distance of the secondary cycles raises the weight of the related error patterns, which yields an increase in the minimum distance of the code. The next section focuses on the influence of the slicing of constituent codes on these cycles and RTZ sequences.

3.4.2 Impact of slicing on the interleaver design

Slicing the constituent codes into distinct trellises facilitates the HI design, since it decreases the number of cycles in the interleaver. Indeed, slicing breaks potential cycles and also suppresses the corresponding RTZ sequences.

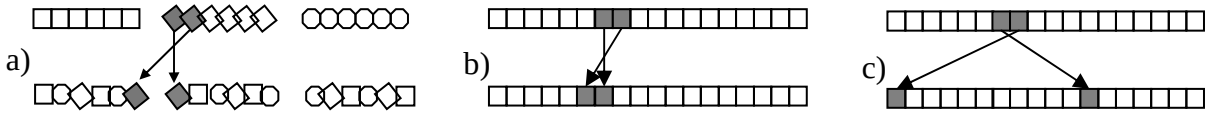


Figure 3-16: Hierarchical interleaver design: a) no cycle for sliced constituent codes (distinct trellises); b) short cycle for conventional constituent codes (single trellis); c) longer cycle with modification of the temporal permutations for conventional constituent codes

Let us analyze the case of the primary cycles. With slicing, it is guaranteed that two symbols of a single slice in one dimension that belong to distinct slices in the other dimension do not form a primary cycle. To illustrate the impact of slicing on primary cycles, let us consider the example of section 3.3.3. The interleaving of two symbols (Figure 3-11.a and Figure 3-11.f) is considered and depicted in Figure 3-16.a. The spatial permutation spreads the first two symbols of the second slice of the natural order to the first and second slices, respectively, of the interleaved order. Then, the temporal permutation puts these symbols in the last and first positions of the first and second slices, respectively. In this case, no cycle exists between the two symbols.

If the same HI is used for an SSTC, the two symbols belong to the same trellis and remain close to each other, as shown in Figure 3-16.b. This configuration considerably degrades the quality of the interleaver. In order to improve it, the temporal permutations should map the symbols at the first and last positions of the sub-blocks, respectively, as shown in Figure 3-16.c. In this case, the temporal permutations and the spatial permutations need to be designed jointly. This joint design methodology suffers from its complexity. With MSTCs,

the spatial and temporal permutations can be designed independently regarding these cycles, which considerably simplifies this task.

Considering the benefits of slicing, another question arises: what is the optimal number of slices to choose? In fact, until now, the number of slices K of MSTCs has been chosen according to the degree of parallelism P of the hardware decoding architecture. But relaxing this constraint, K could be chosen in order to maximize the performance of the turbo code. The next section addresses this question.

3.4.3 Influence of the number of slices for two-dimensional turbo codes

Obviously, the higher the number K of slices, the fewer the number of cycles in the interleaver. On the other hand, as K is increased, the size of each slice is reduced thus rising unwanted side effects. First, the length of the forward and backward recursions becomes small, each recursion being penalized by the absence of known beginning and ending states (due to the use of tail-biting codes). In general, this penalty is negligible in the overall decoding performance, but it becomes visible as the slice size is low and the number of slices is important. In the iterative process, the performance penalty is expressed by a poor convergence behaviour. Second, as the slice size drops, low-weight codewords emerge cancelling the traditional benefits of using RSC codes, which produces high-weight redundancy sequences for low-weight input sequences. For instance, with a circular trellis, a single erroneous symbol produces a number of redundancy bits proportional to the length of the trellis. Therefore, the required minimal distance of the turbo code gives a lower bound on the size of the slices.

In order to study the influence of slicing on the performance of the turbo code, we have simulated the average performance of double-binary MSTCs of rate 1/2 with parameters (1024, 1024/K, K), where the number of slices ranges from 1 to 64, and consequently the frame size ranges from 1024 to 16. A set of 10 random interleavers has been used for this study. In order to take advantage of the slicing property, the interleavers should map symbols of the same slice in one dimension to symbols of different slices in the other one. Therefore, we have chosen to use our two-level hierarchical interleaver HI(K). The spatial permutation is chosen to be a circular shift of amplitude $A(\bar{t})$, whose equation is given by $\Pi_S(t,r) = \left(A(\bar{t}) + r\right) \bmod P$, where $\bar{t} = t \bmod P$ and A is a bijection of variable $\bar{t} \in \{0,...,P-1\}$ to $\{0,...,P-1\}$ chosen at random. The temporal permutations of size M are also chosen at random. The MSTCs (1024, 1024/P, P) are compared to SSTCs of size N = 1024 using the same random two-level hierarchical interleavers HI(K). The resulting turbo code is denoted (1024, 1024, 1) HI(K). Figure 3-17 gives the value of the SNR at which the two turbo codes reach a given FER after 8 iterations of a floating point Max-Log-MAP algorithm. The FER values considered are 10^{-2} and 10^{-3}. In order to lower the influence of the slice size, each slice is decoded using a schedule Σ^{-} with recursions initialized thanks to a pre-processing step of size 32 symbols. All performance curves presented in this manuscript are obtained by Monte-Carlo simulation with a minimum of 25 erroneous frames.

From these curves it is clear that the performance of the MSTC is improved as the number of slices is augmented. When the size gets too small (M = 16 for K = 64), the performance of the MSTC is degraded. The largest improvement is obtained at an FER of 10^{-3}, in the error floor regions of the codes[1]. However, the observed improvement in performance with the number of slices does not come from the slicing but from the interleaver construction that improves the spread of the interleaver. The comparison shows that turbo codes constructed with multiple slices have on average an equivalent performance to SSTCs. Besides, increasing K to 64 leads to a degradation in the MSTC performance that is not observed for the SSTC. The low size of the slice (M = 16) explains this degradation.

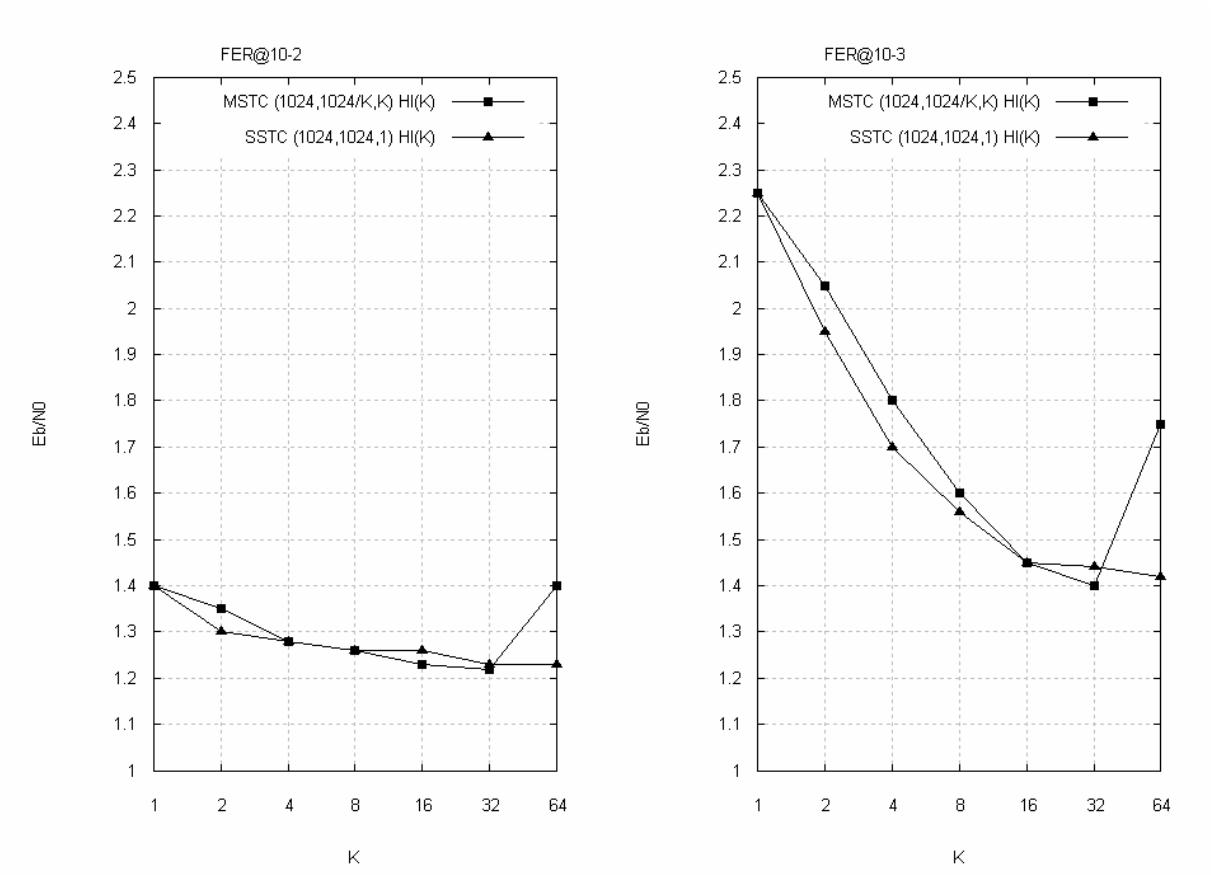


Figure 3-17: Performance of average MSTC (1024, 1024/*K*, *K*) and average SSTC (1024, 1024, 1)

In conclusion, regarding the performance of the code, the number of slices is chosen to be the maximal value that guarantees a minimal size of the slice, which depends on the size N of the frame. We believe, however, that the interest of MSTCs lies in their slicing property facilitating the optimization of a two-level hierarchical interleaver as will be seen in section 3.5.

It is interesting to highlight a particular case. When the number K of slices equals the size of the slices M (ρ = 1) and assuming a bijective spatial permutation, all symbols of one slice in one dimension are distributed into distinct slices of the other dimension. Therefore, no primary cycle exists. We have shown, in Appendix B, that the spread of a MSTC is bounded

[1] Using deterministic and optimized interleavers considerably lowers the error floor obtained here for random interleavers.

by 2*M*. Assuming that the impact of primary cycles on the convergence is predominant, the turbo code has good convergence properties whatever the choice of the spatial and temporal permutations. Furthermore, secondary cycles also have a significant influence on the convergence and the associated SEPs are responsible for the low-weight codewords. Therefore, the question arises as to the possibility of also suppressing secondary cycles with MSTCs. A survey on the necessary conditions on the minimal number of slices has been conducted. It shows that it is possible to suppress all secondary cycles of an interleaver for two-dimensional MSTCs under the condition that the number K of slices is in the order of M^2. Therefore, the size of the frame is as high as M^3. We end up with two contradictory constraints in the search for an optimal value of M. If M is chosen too small, the minimum distance of the code is dominated by single errors in slices. If M is chosen large enough to suppress the influence of single errors, then the size of the frame N is too large and other more complex error patterns are predominant over SEPs. This construction is thus not adapted for practical applications.

Before describing the optimization process of the parallel interleaver we propose two other turbo code design techniques using MSTCs.

3.4.4 Design of three-dimensional Multiple Slice Turbo Codes

In [Gnaedig *et al*-05b], we generalized the design of Multiple Slice Turbo Codes to three-dimensional turbo codes. The key idea is to adapt and optimize the technique of multiple turbo codes to obtain a lower error floor (than for 2-dimensional turbo codes) combined with the parallel decoding property of MSTCs. A novel asymmetric puncturing pattern, has been used to trade off convergence of the code against minimum distance (*i.e.* error floor) in order to adapt the performance of the 3D-MSTC to the requirements of the application. In fact, the two first dimensions of the turbo codes are equally protected and the third dimension is far more punctured and thus less protected. The purpose of the third dimension is to increase the minimum distance of the code sufficiently without sacrificing the convergence. An adapted heuristic optimization method of the interleavers has been described, leading to excellent performance. It involves a two step approach: first an optimization of the first permutation between the first two equally protected dimensions is performed, then, the second permutation is defined in order to maximize the minimum distance of the code.

With the asymmetry of the puncturing pattern, it has also been shown that the decoding of the third dimension (*i.e.* the most punctured) should be idled during the first decoding iterations. From this observation, two new adapted iterative decoding structures have been proposed, which reduce the complexity of the decoder. More details on this construction are given in Appendix D.

3.4.5 Generalized Multiple Slice Turbo Codes

Using slices, the notion of first and second dimension can be relaxed: the code can be seen as an LDPC-like code where the parity check constraints are simply replaced by CRSC codes. Tan example of a corresponding Tanner graph [Tanner-81] is presented in Figure 3-18. In this

graph, the information and redundancy symbols of one elementary code (a slice) are connected through the edges to the symbols of the codeword. Four slices are used to build a code of length $n_c = 12$ and dimension $k_c = 8$.

This flexible representation makes it possible to build parallel concatenations of CRSC codes, and also serial concatenations. Although this generalized code construction seems attractive, we have not so far exploited its possibilities.

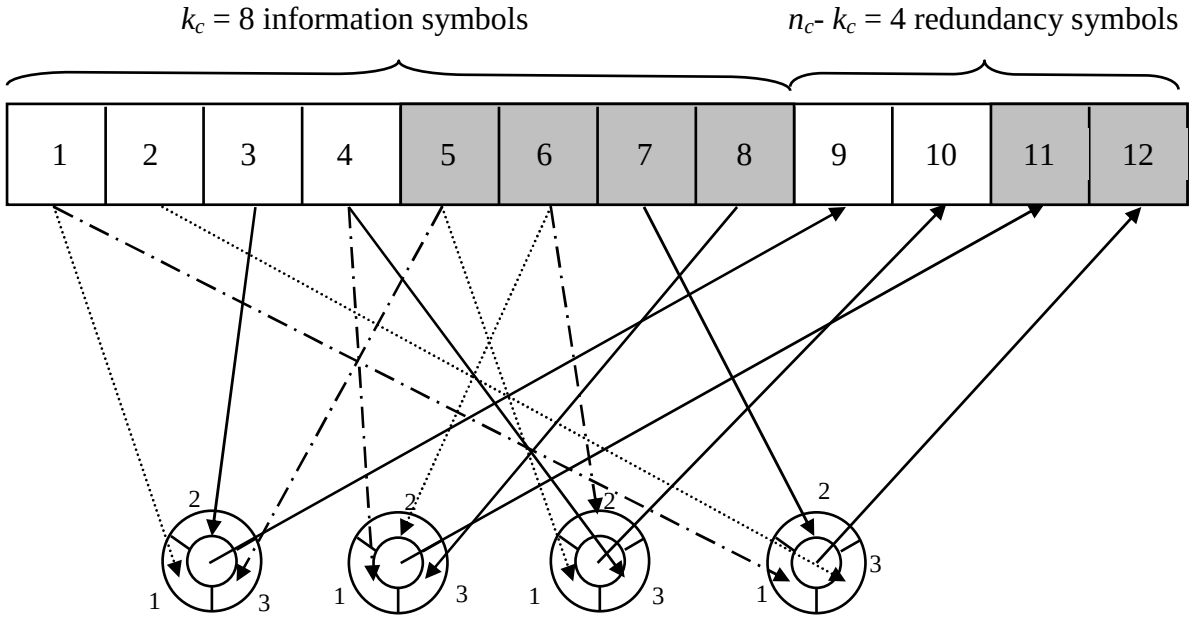


Figure 3-18: Tanner graph of a Generalized Multiple Slice Turbo Code

3.5 Hierarchical interleaver optimization for Multiple Slice Turbo Codes

This section aims to describe the optimization steps of the two-level hierarchical interleaver design with MSTCs. It uses the slicing properties of MSTCs in the case $\rho \geq 2$, *i.e.* two slices of the natural and interleaved order have at least two symbols in common. The optimization process of the interleaver is based on three steps: optimization of the temporal permutation, optimization of the spatial permutation, and finally, a refinement step of the temporal permutation is performed by addition of a local disorder. This process will then be validated by simulation.

3.5.1 Temporal permutation

The aim of temporal permutation is to maximize the minimum spread in order to eliminate low-weight PEPs and to reduce the correlation between the extrinsic information inputs.

As a first step, a simple regular temporal permutation is chosen in order to simplify the study of the interaction between the temporal and the spatial permutations. In this sub-section, it is optimized by analyzing primary cycles only. Next, in sub-section 3.5.3, more irregularity will be introduced in the temporal permutation in order to improve the performance further. The temporal permutation used in this study is given by:

$$\Pi_T(t) = \lambda \cdot t \bmod M, \quad \textbf{(3-5)}$$

where λ is an integer, and where λ and M are mutually prime. This choice is justified by the hardware simplicity of the temporal generator and the good performance that can be obtained. The only parameter to be optimized is then λ.

Since the spatial permutation is P-periodic and bijective, two symbols of a same slice having a distance which is not a multiple of P in the second dimension are not in the same slice in the first dimension. Their spread is then infinite. Reciprocally, two symbols (k_1, k_2) of the second dimension that belong to a primary cycle have a distance verifying $\|k_1 - k_2\|_M = |q \cdot P|_M$, where q =1,…, ρ - 1. Their temporal indices are denoted t_1 and t_2 respectively, and thus $|t_1 - t_2|_M = |q \cdot P|_M$. With the temporal permutation (3-5), we obtain:

$$\|\Pi(k_1) - \Pi(k_2)\|_M = |\lambda \cdot t_1 - \lambda \cdot t_2|_M = |\lambda \cdot q \cdot P|_M \quad \textbf{(3-6)}$$

Thus, there are ρ possible values for the spread between two symbols. Thus, the overall minimum spread of the interleaver is given by:

$$S = \min_{q=1,\ldots,\rho-1} \left(|q \cdot P|_M + |\lambda \cdot q \cdot P|_M \right) \quad \textbf{(3-7)}$$

The parameter λ is then chosen to maximize the spread S. Since $\lambda < M$, an exhaustive search is performed in order to obtain the highest spread. The temporal permutation also has an influence on the internal secondary cycles and on the corresponding SEPs. These cycles will be studied in section 3.5.3.

3.5.2 Spatial permutation

The choice of spatial permutation is made by analyzing its influence on external secondary cycles and their associated error patterns. Hence, in the following, we will implicitly refer to external cycles when speaking about secondary cycles. Note that the summary distance of secondary cycles does not increase with high spread, and therefore the weight of SEPs does not improve with high spread. For the temporal permutation given in (3-5), the spatial permutation aims to maximize the summary distance of the secondary cycles.

To simplify hardware implementation, the spatial permutation is constrained to be a circular shift of amplitude $A(\bar{t})$, whose equation is given by:

$$\Pi_S(t, r) = \left(A(\bar{t}) + r\right) \bmod P \quad \textbf{(3-8)}$$

where $\bar{t} = t \bmod P$ and A is a bijection of variable $\bar{t} \in \{0,\ldots,P-1\}$ to $\{0,\ldots,P-1\}$.

In order to obtain a graphical representation of the secondary cycles, the spatial permutation is represented by a square matrix **T** of size P. For this matrix, the i^{th} row is associated with the i^{th} slice of the second dimension and the j^{th} column is associated with the j^{th} slice of the first dimension. As defined at the end of section 3.3.2, **T**(i,j) is the temporal index verifying $\Pi_S(\mathbf{T}(i, j), i) = j$, *i.e.*, it represents the set $\tau_{i,j}$ of the ρ symbols of the i^{th} slice of the interleaved order that also belong to the j^{th} slice of the natural order.

Let us now design the matrix **T** corresponding to equation (3-8). For fixed $\bar{t}$, the ρ symbols in slice r = 0 of the second dimension correspond to symbols from slice $A(\bar{t})$ of the first dimension. This is expressed mathematically as $\mathbf{T}(0, A(\bar{t})) = \bar{t}$. Moreover, for fixed $\bar{t}$, $\Pi_S(\bar{t}, s)$ is a rotation. This means that the ρ symbols in slice r = 1 correspond to symbols from slice $(A(\bar{t}) + 1) \bmod P$, and thus, $\mathbf{T}(1, (A(\bar{t}) + 1) \bmod P) = \bar{t}$. More generally, for i = 0,…, P-1, $\mathbf{T}(i, (A(\bar{t}) + i) \bmod P) = \bar{t}$. Hence the P sets τ_i of ρ temporal indices form a circular diagonal $d_{\bar{t}}$ in matrix T (see Figure 3-19), which is then a Toeplitz matrix. The matrix T of the example in section 3.3.3, where $A(\bar{t}) = \{0,2,1\}$ is given by (3-9):

$$\mathbf{T} = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{bmatrix}. \qquad \textbf{(3-9)}$$

$$T = \begin{array}{c} \begin{array}{ccc} d_0 & d_2 & d_1 \end{array} \\ \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 3-19: Representation of the circular diagonals d_0, d_1, d_2 for matrix T

For $\bar{t} = 0$, the symbols in slices 0, 1 and 2 of the second dimension correspond to the symbols in slices 0, 1, 2 of the first dimension, respectively (this diagonal d_0 corresponds to $A(\bar{t})$ = 0). Then for $\bar{t} = 1$, the symbols in slices 0, 1 and 2 of the second dimension correspond to the symbols in slices 2, 0, 1 of the first dimension (this diagonal d_1 corresponds to $A(\bar{t})$ = 2). Finally, for $\bar{t} = 2$, the symbols in slices 0,1 and 2 of the second dimension correspond to the symbols in slices 1, 2 and 0 of the first dimension (this diagonal d_2 corresponds to $A(\bar{t})$ = 1).

A secondary cycle is represented in the spatial permutation matrix T by a rectangle (i_1, j_1, i_2, j_2) as shown in Figure 3-20. Note that **T**(i,j) represents the set $\tau_{i,j}$ of ρ distinct symbols. Thus the rectangle in Figure 3-20 represents ρ^4 distinct secondary cycles. For the sake of simplicity, we consider only one of these secondary cycles, which does not change the results of our study. The temporal indices in the second dimension are denoted $t_{i_1 j_1}$, $t_{i_1 j_2}$ for the first slice and $t_{i_2 j_1}$, $t_{i_2 j_2}$ for the second slice. Let $\Pi_T(t_{i_1 j_1})$, $\Pi_T(t_{i_1 j_2})$, $\Pi_T(t_{i_2 j_1})$ and $\Pi_T(t_{i_2 j_2})$ be their corresponding indices in the slices of the first dimension. The summary distance of this secondary cycle is given by

$$d_{sum} = \left|t_{i_1 j_1} - t_{i_1 j_2}\right|_M + \left|t_{i_2 j_1} - t_{i_2 j_2}\right|_M + \left|\Pi_T(t_{i_1 j_1}) - \Pi_T(t_{i_2 j_1})\right|_M + \left|\Pi_T(t_{i_1 j_2}) - \Pi_T(t_{i_2 j_2})\right|_M \quad \textbf{(3-10)}$$

With the example of Figure 3-20, $t_{i_1 j_1} = 2$, $t_{i_1 j_2} = 1$, $t_{i_2 j_1} = 0$, $t_{i_2 j_2} = 2$ and $\Pi_T(t_{i_1 j_1}) = 4$, $\Pi_T(t_{i_1 j_2}) = 3$, $\Pi_T(t_{i_2 j_1}) = 1$, $\Pi_T(t_{i_2 j_2}) = 3$. The summary distance of this cycle is $d_{sum} = 6$.

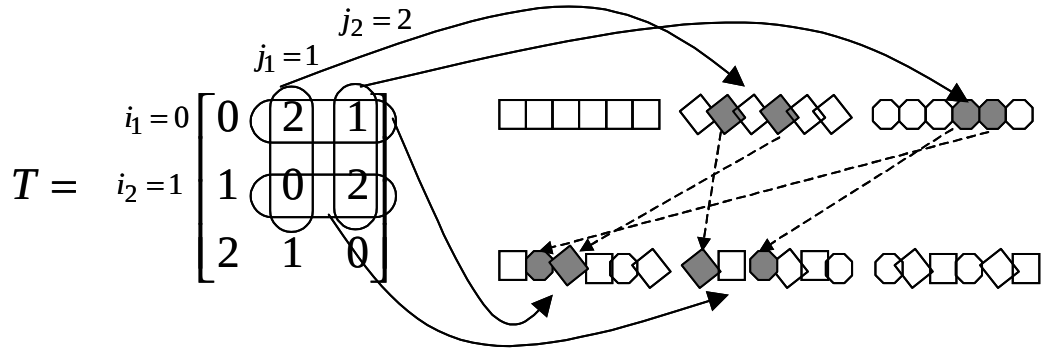


Figure 3-20: External secondary cycle representation on the spatial permutation: (i_1, j_1, i_2, j_2) = (0, 1, 1, 2) and $\Pi_T(t)$={1,4,3,2,5,0}

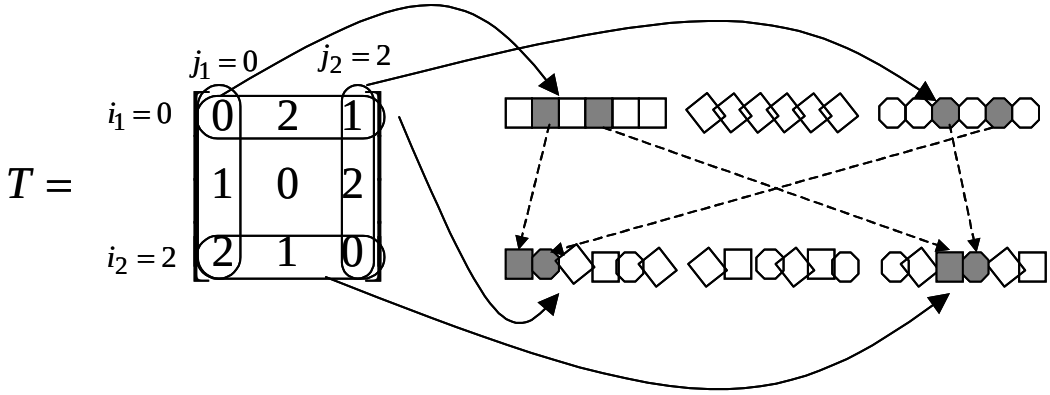


Figure 3-21: External rectangular secondary cycle representation on the spatial permutation (i_1, j_1, i_2, j_2) = (0,0,2,2), and $\Pi_T(t)$={1,4,3,2,5,0}

Theorem 3-1: For temporal permutation (3-5) and spatial permutation (3-8), if an external secondary cycle verifies $\left|t_{i_1 j_1} - t_{i_1 j_2}\right|_M = \left|t_{i_2 j_1} - t_{i_2 j_2}\right|_M = \Delta t$ then it is rectangular and its summary distance is given by $S_d = 2 \cdot \left(\Delta t + \left|\lambda \cdot \Delta t\right|_M\right)$.

In fact, using the temporal permutation (3-5), $\left|t_{i_1 j_1} - t_{i_1 j_2}\right|_M = \left|t_{i_2 j_1} - t_{i_2 j_2}\right|_M = \Delta t$ implies $\left|\Pi_T(t_{i_1 j_1}) - \Pi_T(t_{i_2 j_1})\right|_M = \left|\Pi_T(t_{i_1 j_2}) - \Pi_T(t_{i_2 j_2})\right|_M = \left|\lambda \cdot \Delta t\right|_M$.

Note that a summary distance S_d of an external rectangular secondary cycle is twice the spread between two symbols distant from Δt. But unlike equation (3-7), Δt is no longer necessarily a multiple of P. Hence, an inappropriate choice of the spatial permutation may lead to rectangular secondary cycles with low summary distances and to low SEPs. For example, with the regular permutation $A(\bar{t}) = \{0,1,...,P-1\}$, all secondary cycles are

rectangular. This case is the worst since any value of Δt leads to a rectangular secondary cycle. The bijection *A* should be chosen as "irregular" as possible in order to minimize the number of rectangular secondary cycles. Since an exhaustive search for the best possible permutation is computationally too high, empirical rules have to be derived. Usually, the irregularity of an interleaver is characterized by its dispersion [Heegard *et al*.-99]. We introduce a new appropriate definition of dispersion.

Let us first define the displacement vector $(\Delta\bar{t},\Delta A)=\left(\left|\bar{t}_2-\bar{t}_1\right|_P,\left|A(\bar{t}_2)-A(\bar{t}_1)\right|_P\right)$ for any couple of diagonals $d_{\bar{t}_1}$ and $d_{\bar{t}_2}$. Due to the *P*-periodicity of the spatial permutation, $\Delta\bar{t}$ is defined with distance $\min\left(\left|\bar{t}_2-\bar{t}_1\right|,P-\left|\bar{t}_2-\bar{t}_1\right|\right)$. Similarly, due to the *P*-periodicity of the spatial permutation, ΔA is also defined with the $\left|\ \right|_P$ distance. Indeed, for each distance, there are two possible configurations. For example, in Figure 3-19, on the first line, diagonals d_0 and d_1 are separated by two positions but by only 3-2 = 1 position on the second line. For the sake of unicity of the displacement vector, we consider only the one with smallest distance $\left|\ \right|_P$. We define the set of displacement vectors as:

$$D(A)=\left\{(\Delta\bar{t},\Delta A),\Delta\bar{t}=\left|\bar{t}_2-\bar{t}_1\right|_P,\Delta A=\left|A(\bar{t}_2)-A(\bar{t}_1)\right|_P,(\bar{t}_1,\bar{t}_2)\in\{0,\ldots,P-1\}^2\right\} \qquad \textbf{(3-11)}$$

The dispersion of the spatial permutation is then defined as $D=\left|D(A)\right|$ the cardinal of the set of displacement vectors. When $D(A)$ contains two displacement vectors $(\Delta\bar{t},\Delta A)_1=(\Delta\bar{t},\Delta A)_2$, this means that there exists a secondary cycle which verifies $\left|t_{11}-t_{12}\right|_M=\left|t_{21}-t_{22}\right|_M$. From Theorem 1, this secondary cycle is "rectangular". Hence, the higher the dispersion, the fewer the number of "rectangular" secondary cycles. The choice of bijection *A* is made by maximizing the dispersion *D*. Its maximization reduces the number of rectangular secondary cycles, increases the minimum summary distance among all secondary cycles, and introduces randomness in the interleaver.

We have seen the influence of spatial permutation on secondary cycles and secondary error patterns. Other error patterns that are combinations of secondary and primary cycles also benefit from their irregularity. Thanks to this optimization, both the convergence of the code and its minimum distance are increased compared to a regular spatial permutation.

3.5.3 Further optimization

With increasing frame size, the study of PEPs and SEPs alone is not sufficient to obtain efficient interleavers. Indeed, other error patterns appear, penalizing the minimum distance. In practice, the analysis and the exhaustive counting of these new patterns are too complex to be performed. Moreover, the temporal permutation (3-5) is too regular and leads to many internal rectangular secondary cycles. Indeed, for an internal secondary cycle as depicted in Figure 3-13.d), 4 symbols belong to the same slice in the second dimension. Since the spatial permutation is *P* periodic, the distances between the temporal indices of any two couples are necessarily equal in the second dimension. Moreover, they are equal in the first dimension as

well. These cycles may lead to low-weight SEPs. Thus, to increase minimum distance, the temporal permutation given by (3-5) is modified like for the ARP interleaver, which has been thoroughly explained in [Berrou *et al*.-04]. This optimization step involves introducing a local disorder in the regular temporal permutation (3-5). Every permuted index $\Pi_T(t)$ is shifted modulo *M* of an amplitude that is not the same for all indices *t*. This is achieved by adding four coefficients $(\mu(i))_{0\le i<4}$ (3-12) to the temporal permutation. These coefficients are lower than or equal to *M* and verify that their values modulo 4 are all different.

$$\Pi_T(t) = \lambda \cdot t + \mu(t \bmod 4) \bmod M \qquad \textbf{(3-12)}$$

The parameter λ is the same as the one chosen in sub-section 3.5.1. Clearly, this local disorder reduces the overall spread of the interleaver that has been optimized with the choice of λ in the regular temporal permutation. But an interleaver not necessarily having a maximal spread can also achieve good performance and might even improve it at low error rates. Moreover, the local disorder modifies the summary distance of the secondary cycles: rectangular secondary cycles may not remain rectangular and vice-versa. Nevertheless, this local disorder introduces more irregularity in the interleaver and may lead to a better interleaver with higher minimum distance and better convergence. The optimal coefficients $(\mu(i))_{0\le i<4}$ are those leading to the highest minimum distance. An exhaustive evaluation of the minimum distance for all possible parameters can be achieved by using the Error Impulse Method (EIM) proposed by Berrou *et al.* [Berrou *et al*.-02].

Since the local disorder is of period 4, the size of the slice should be a multiple of 4 in order to design a homogeneous code with even protection for all symbols of a slice. Hence for a double-binary code, the size of the slice has byte (8-bit) granularity. In addition, for an (*N*,*M*,*P*) turbo code, the granularity of the frame size is *P* bytes.

In this section, the interleaver design has been discussed and its optimization is performed in three steps: maximization of the spread through a regular temporal permutation, maximization of the dispersion of the spatial permutation and introduction of a local disorder in the temporal permutation to increase the randomness of the interleaver. This optimization methodology resulting from the study of cycles in the interleaver and of the RTZ sequences is a major contribution of this thesis. It is validated in the next section by simulation.

3.5.4 Validation of the optimization process

Applying the different methods developed in the preceding sections, a (2048, 256, 8) MSTC is designed using 8-state double-binary CRSC codes with parameters (8, 2, 3). An intra-symbol permutation as defined in chapter 1.3 is also applied to increase the minimum distance as in the DVB-RCS standard [DVB-RCS]. These rate 1/2 codes are compared in Figure 3-22 with an SSTC (2048, 2048, 1), constructed with one slice using an ARP interleaver with parameters $\lambda = 45$ and $(\mu(i))_{0\le i<4} = \{0, 1060, 52, 1040\}$. The latter is also optimized with the same strategy as for the multiple slice turbo code: a large number of interleaver parameters are tested using the EIM. This optimization leads to an evaluated

minimum distance of 20. Let us illustrate the effects of the different optimization steps of the interleaver. First, only the temporal permutation is optimized, assuming a regular spatial permutation (curve TO, in Figure 3-22). The minimum weight of the PEP is then above 30 but the SEP introduces a low minimum distance of 14 for the code. After optimizing the spatial permutation, low-weight SEPs are eliminated and the minimum distance increases to 18 (curve SO). Then, when the local disorder is added to the temporal permutation, the minimum distance is raised to 21 (curve LD).

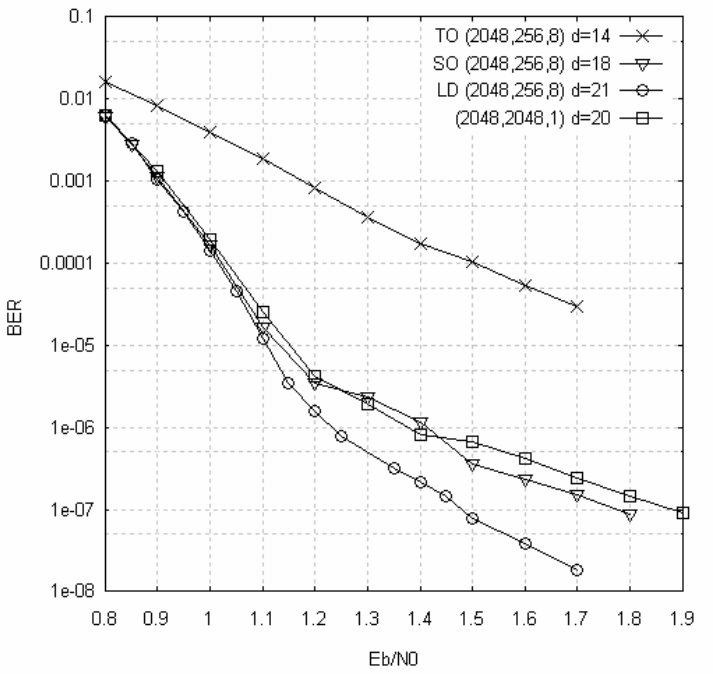


Figure 3-22: Performance of the (2048, 256, 8) and (2048, 2048, 1) double-binary turbo codes for 8 iterations of the Log-MAP algorithm (L = 128, w_d = 6)

The performance curves are obtained with the Log-MAP algorithm with a window size L = 128 and w_d = 6. The results are in accordance with the evaluated minimum distances of the codes. The (2048, 256, 8) code performs 0.3 dB better at a BER of 10^{-7} than the SSTC (2048, 256, 1). Nevertheless, there is less flexibility in choosing the frame size for the MSTC. Indeed, due to the period 4 in the ARP interleaver, the size of the slice has byte granularity. Thus the granularity of the MSTC is a multiple of the number of slices, whereas the SSTC code has byte granularity.

3.6 Performance of Multiple Slice Turbo Codes

In this section, we present the performance of several MSTCs designed with various frame sizes and number of slices.

3.6.1 Influence of the code rate and the frame size

In this sub-section we extend the results obtained for rate 1/2 to higher codes rates and for an information block size of N = 1024 double-binary symbols. Figure 3-23, Figure 3-24 and Figure 3-25 compare the BER and FER performance of MSTCs constructed with 8 slices to SSTCs associated with ARP interleavers for rates 1/2, 2/3 and 4/5, respectively. The

minimum distance of these codes evaluated with the EIM are also given. The performance curves are obtained with 8 iterations of the Log-MAP algorithm with a window size $L = 128$ and $w_d = 6$. Two frame sizes are used: $N = 2048$ and $N = 1024$ (only rates 1/2 and 2/3 are considered for this latter frame size). The simulated performance shows that the gain of the MSTC decreases as the code rate increases, and also as the frame size decreases. For example, with $N = 2048$, for a target FER of 10^{-4} the FER of the MSTC shows a gain of about 0.2 dB, 0.1 dB and 0.05dB for rates 1/2, 2/3 and 4/5, respectively, compared to the SSTC. For $N = 1024$, the gain is reduced to less than 0.1 dB. Hence, for shorter information block lengths, the two turbo codes have the same performance in terms of convergence and minimal distance. The larger gain for longer block lengths can be explained by the good interleaver design for MSTCs and especially its higher irregularity. Indeed, the spatial permutation of the code with 8 slices introduces another degree of freedom, *i.e.* irregularity, into the interleaver design, which is beneficial for longer block sizes.

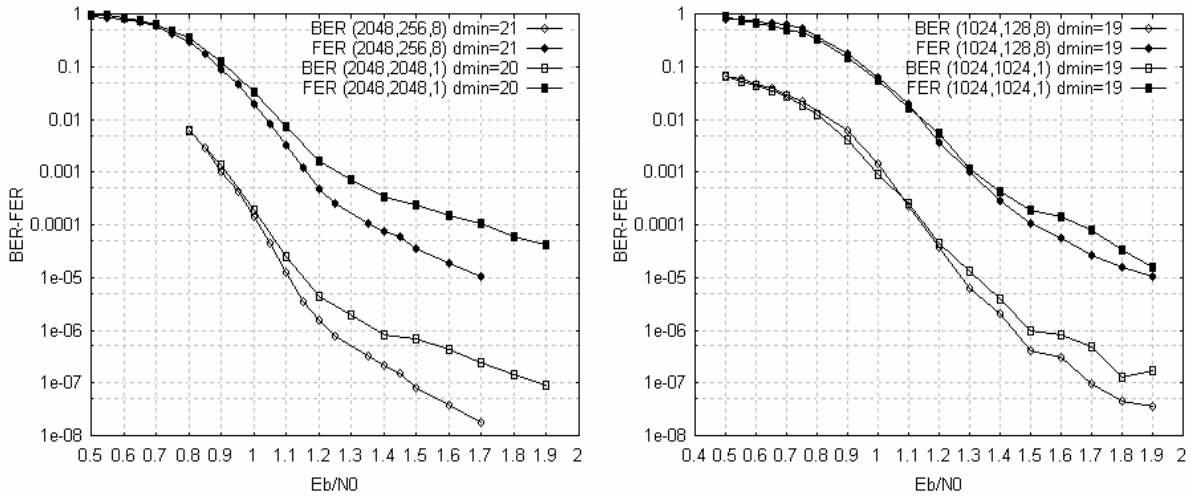


Figure 3-23: Performance comparison of the (2048, 256, 8) and (1024, 128, 8) MSTCs to the (2048, 2048, 1) and (1024, 1024, 1) SSTCs for *R*=1/2

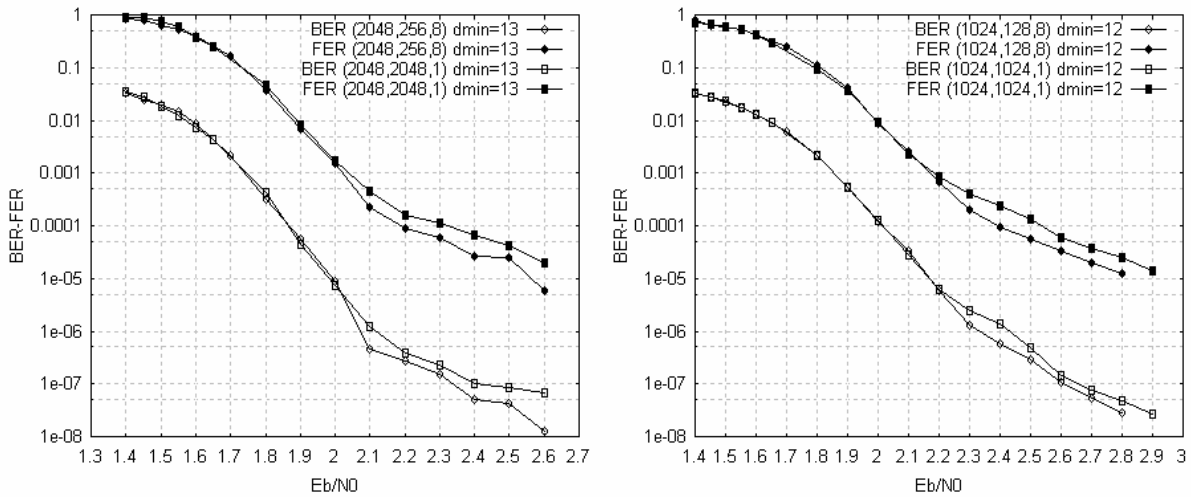


Figure 3-24: Performance comparison of the (2048, 256, 8) and (1024, 128, 8) MSTCs to the (2048, 2048, 1) and (1024, 1024, 1) SSTCs for R=2/3

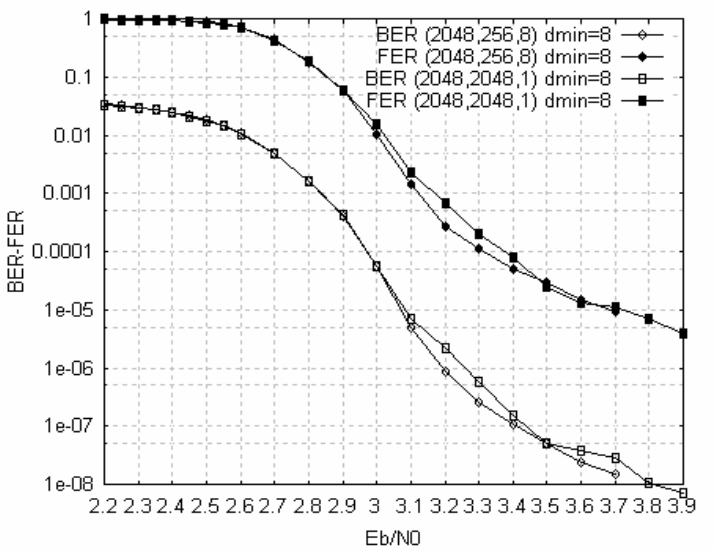


Figure 3-25: Performance comparison of the (2048, 256, 8) MSTC to the (2048, 2048, 1) SSTC for R=4/5

3.6.2 Performance of other MSTCs

Figure 3-26 shows the performance of the (11400, 760, 15) MSTC, simulated with 10 iterations of the Log-MAP algorithm with L = 128 and w_d = 6. Note that this MSTC was used in TurboConcept's proposal for introducing a turbo coding scheme into the DVB-S2 standard in January 2003. Basically, this turbo code was concatenated with an outer Reed-Solomon code (see [Huffman *et al*.-03]) to lower the error floor of the turbo code.

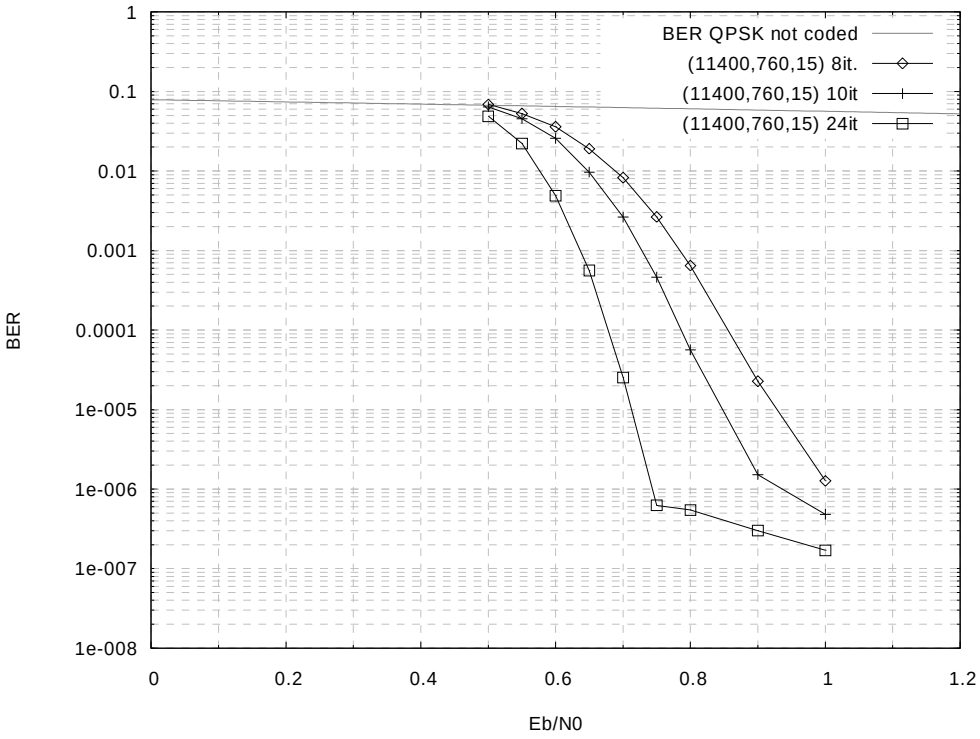


Figure 3-26: BER performance of the rate 1/2 (11400, 760, 15) MSTC

Performance of other MSTCs for various frame sizes and code rates are presented in Appendix C.

3.7 Other structured interleavers for parallel decoding

The parallel architecture presented in the second section of this chapter can support other constructions of interleavers, associated with their respective memory mapping. This includes hierarchical and also non-hierarchical interleaver constructions, described in the following sub-sections.

3.7.1 Hierarchical interleaver design

Other hierarchical interleaver designs enabling parallel turbo decoding and resolving the problem of accessing the memory banks concurrently have been proposed in the literature.

Nimbalker *et al.* designed a parallel hierarchical interleaver using the same principle as the interleaver construction used for MSTCs [Nimbalker *et al*.-03] [Nimbalker *et al*.-04]. This interleaver is used with a conventional parallel turbo code (*i.e.* without slices) and an appropriate design methodology is proposed. The temporal permutation is fixed and corresponds to a bit-reversal permutation. The performance of the turbo code is optimized by the selection of a spatial permutation breaking the cycles in the interleaver and increasing the minimum distance of the code. Binary turbo codes are designed and show equivalent performance to the turbo code of the 3GPP standard.

In the same way, Kwak *et al.* designed a parallel interleaver in [Kwak *et al*.-03], which relies on the separation of the interleavers into two separate permutations: a temporal and a spatial permutation. No interleaver optimization techniques are addressed in this paper, dealing mainly with parallel turbo decoding.

3.7.2 Non-hierarchical interleaver design

Other constructions of interleavers have been proposed that do not rely on a direct transformation of the parallel decoding structure to the interleaver structure.

In [Weiß *et al*.-01], a code construction using CRSC codes was used in conjunction with a line-column interleaver. Line-column interleavers have been used successfully for Turbo Product Codes using linear constituent codes. Likewise, the information symbols are stored in a matrix, whose rows and columns are encoded by CRSC codes. The turbo code construction is optimized by an appropriate choice of the generator polynomials of the convolutional codes, enabling the minimum distance of the code to be increased. The memory mapping maps each circular diagonal of the interleaver matrix into a distinct memory bank. Like for TPC, this memory mapping allows parallel decoding of the rows and the columns. Because of the fixed interleaver, this construction is not flexible. In addition, an increase in the minimum distance can only be achieved by changing the constituent convolutional codes, which might result in a higher decoding complexity. Although this construction was not devised for parallel decoding, thanks to the block interleaver, it allows the parallel decoding of the rows or of the columns.

Similarly, in [Giulietti *et al.*-02a], Giulietti *et al.* developed a systematic method to generate collision-free interleavers based on block interleavers. The information symbols are written into the permutation matrix row by row, and read out column by column. The decoder architecture uses a straightforward memory organization, mapping each row of the matrix into one memory bank, and a dedicated processor for decoding each row. The rows of the matrix can therefore be decoded simultaneously without collisions. In the interleaved order, each processor decodes the symbols corresponding to one column. In order to avoid collisions, cyclic shifts of the columns are applied to generate a collision free interleaver. Performance of the code can be improved with additional intra-row and inter-row permutations.

Another approach for designing interleaver for parallel decoding was proposed by Berrou *et al.* in [Berrou *et al*.-04]. In fact, the structure of the interleaver was devised regarding two criteria: the related performance of the turbo code and the ability to be generated on the fly by simple generation rules. In addition, the properties of this structure can be used to decode such a turbo code in parallel. An Almost Regular Permutation (ARP) is based on a regular permutation with additional disorder of permutation cycle C_p. Then the interleaved index is given by:

$$\Pi(k) = \lambda . j + \mu(k \bmod C_p) + 1 \bmod N \qquad \textbf{(3-13)}$$

The parameter λ is chosen in order to maximize the spread of the interleaver, and is chosen to be an integer relatively prime with N close to $\sqrt{2N}$. The parameters μ are chosen as multiples of C_p. They introduce a local disorder in the permutation thus increasing its dispersion. They are chosen in order to maximize the asymptotic performance of the code evaluated using for instance the EIM.

Because of the C_p -periodicity of the interleaver, the turbo code can be decoded in parallel by C_p processors. For this parallel architecture, the memory organization maps all symbols with the same congruence modulo C_p in a single memory bank. Then, the C_p portions of the circle representing the N-stage trellis are decoded simultaneously by the C_p processors. At the same time, the processors deal with data corresponding to the C_p possible congruence (see Figure 3-27). This constraint means that the first processor starts with an address with congruence 0, the second with congruence 1, and so on. During the next clock period, the first processor handles data with congruence 1, the second with congruence 2, etc. This schedule can be performed both in the natural and interleaved order, because the interleaver maintains the class of congruence structure.

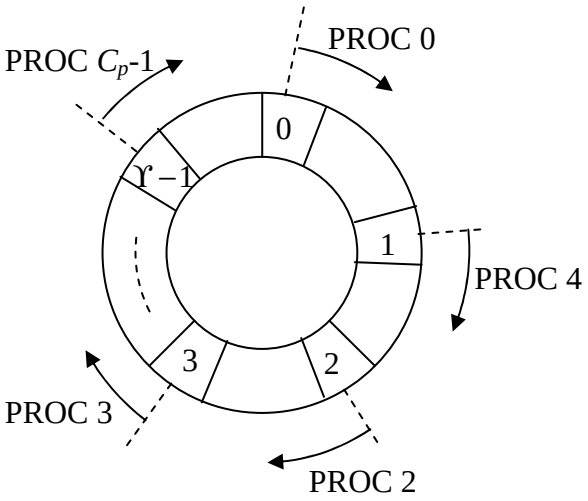


Figure 3-27: Parallel decoding of turbo codes using an ARP interleaver

ARP interleavers can be implemented on the interleaver datapath described in the second section in a similar manner as other types of interleavers described throughout this chapter. The Spatial Permuter and Memory Address Generators are obtained easily from the interleaver equation (3-13). Moreover, it has been discovered recently in [Berrou *et al*.-04] that if the frame size N is a multiple of $p \cdot C_p$, then $p \cdot C_p$ parallel processors can be used in conjunction with $p \cdot C_p$ memory banks storing the classes of congruence modulo $p \cdot C_p$. This observation is very important since ARP interleavers can be constructed with a great flexibility, without taking care of the parallel decoding architecture. Actually, the permutation cycle C_p is chosen to be a constant optimized for performance (generally 4), which still allows a rather large range in the choice of the number of processors.

3.8 Conclusion

In this chapter we have addressed the resolution of the problem of concurrent accesses to the memory resulting from the parallel decoding of turbo codes with multiple processors.

In the first step, we have classified the solutions solving the main parallel decoding problem with respect to the stage at which they consider the problem.

- At the execution stage: based on a shuffle network this solution operates during the execution of the parallel decoder to treat concurrent accesses.
- At the compilation stage: this solution involves finding a memory mapping matched to the interleaver and to the Forward Backward schedule, thus guaranteeing the absence of conflicts. This "compilation" can be achieved for every interleaver and results in a parallel interleaver that can be implemented easily with three layers of permutations, resulting in a low complexity shuffle network.
- At the design stage: our contribution involves the design of the interleaver not only for optimizing the error decoding performance of the coding scheme, but also for the

suitability offered to the code to be decoded in parallel iteratively, by solving *a priori* the problem of concurrent accesses.

While execution and design-stage solutions have appeared almost simultaneously, the compilation-stage solution has been re-discovered more recently (similar techniques had been used in the general area of parallel architectures, and were re-invented in the context of parallel turbo decoding). The execution-stage solution suffers from the additional hardware dedicated to the implementation of the shuffle network, whereas the compilation-stage solution suffers from the memory requirement to store the memory mapping. For this reason, this latter solution is more adapted for turbo codes with reasonable frame sizes. For long frame size, the complexity of the memory storing the mapping memory becomes prohibitive, and the first solution is preferable. The two solutions have the important advantage of being applicable whatever the code and the interleaver. They are of particular interest for standardized or existing turbo codes. On the other hand, when the code is not fixed, the most promising solution to reduce further the complexity of the parallel interleaver involves a joint interleaver-architecture design. We have shown that the introduction of a structured interleaver enables us to find easily a memory mapping preventing any parallel conflict and that can be implemented with low complexity.

In the first step, we have described a general parallel architecture with P memory banks and P processors along with the parallel interleaver datapath, which is composed of two levels: memory address generators for shuffling data inside the memory banks, and a Spatial Permuter shuffling the data between the memory banks. A memory mapping of the data into the memory bank is also associated with this architecture and with the interleaver. Several interleaver constructions satisfying this interleaver structure have been described throughout this chapter.

Our solution is based on an original family of convolutional turbo codes adapted to this architecture: Multiple Slice Turbo Codes (MSTCs). This family uses several CRSC codes in each dimension and a hierarchical interleaver design, which is mapped directly onto the parallel datapath structure thanks to spatial and temporal permutations, allowing a very simple and elegant architecture for turbo decoders. The impact of slicing the turbo code performance has been addressed and led to the conclusion that, on average, the performance is equivalent to that of a single-slice turbo code (SSTC). The gain offered by this family of turbo codes is provided by the structure of its hierarchical interleaver.

Then, the optimization of the interleaver has been presented, which is carried out in three steps: high spread through the regular temporal permutation, maximization of the dispersion of the spatial permutation and introduction of a local disorder in the temporal permutation to increase the randomness of the interleaver. The performance simulations show that the parallelism constraint in the interleaver construction introduces no degradation in performance, and a good interleaver construction can even improve it. For this reason, a solution based on MSTCs was proposed to the DVB-S2 standardization committee.

Finally, we have also described other structured interleavers based either on a hierarchical interleaver design, or on a structure suitable for parallel decoding. The most promising interleaver structures are ARP interleavers, because of their flexibility and performance. Actually, these interleavers can be generated solely under the performance constraint, independently of the hardware architecture, and their inherent structure makes them suitable for flexible parallel decoding. From a historical point of view, when we proposed MSTCs, ARP interleavers were already known, but they had a limited degree of parallelism of 4. It was discovered only recently that under some conditions the same interleaver structure allows more than 4 processors to work in parallel. Nevertheless, the properties of slicing and the hierarchical interleaver design will be used efficiently in the next chapter, showing an important benefit of this turbo code family. In particular, slicing the constituent codes will not be used to increase the performance of the code but to increase the activity of the corresponding architecture. Moreover, we also mention that MSTCs are of great interest for the analog decoding of turbo codes [Arzel *et al.*-05].

Chapter 4

Increasing the activity of parallel architectures

Contents:

Chapter 4. Increasing the activity of parallel architectures

In the second chapter, we presented the parallel architecture as a means to improve the throughput of the turbo decoder. The first problem related to parallel decoding - the collisions resulting from concurrent accesses to the memory – has been tackled in the previous chapter. However, increasing the degree of parallelism of an architecture does not necessarily imply a linear increase of the throughput. As seen in chapter 2.3, this remark is especially crucial for parallel turbo decoder architectures, as increasing the number of parallel processors leads to a reduction in their activities. This chapter explores solutions for retrieving high activity. This research field and the techniques derived hereafter represent an original contribution of this thesis. To the best of our knowledge, this problem has not yet been addressed in the literature.

The first section describes a solution involving the design of a schedule adapted to parallel decoding. By optimizing schedule SW-Σ^0, an activity increase is achieved through a sharing of computational units between two processors. The utilization of this schedule in the context of parallel decoding is one of our contributions for increasing the activity. Although the activity improvement is significant, it is not always sufficient and other solutions are studied.

In the second section we investigate the association of parallel and serial decoding in the same decoder. We will show that our proposition of a hybrid architecture imposes constraints at the system level, which are not always acceptable

The third solution discussed introduces a modification of the decoding schedule. Instead of a conventional processing the half-iterations strictly sequentially, which as the side effect to reduce activity, we propose a new schedule using an overlapping in the processing of consecutive half-iterations. However, we will show that this processing overlapping leads to another type of memory conflict, which we call consistency conflict: the extrinsic data required at the beginning the current half-iteration might not yet have been produced by the previous half-iteration. After evaluating the average number of consistency conflicts for a uniform interleaver as a function of the overlapping depth, we will conclude on the necessity for appropriate solutions to tackle this problem. Similarly to the solutions developed in chapter 3.1 for handling parallel conflicts, consistency conflicts can be treated at either the execution-stage, the compilation stage or the design stage.

In the fourth section, we present a technique and the associated hardware architecture handling consistency conflicts during the execution of the algorithm. The performance of this technique is also evaluated.

In the fifth section, we develop a promising solution based on a joint interleaver-architecture design adapted to the parallel schedule and preventing any consistency conflicts.

This latter alternative uses the properties of the hierarchical interleaver associated especially with MSTCs to suppress all consistency conflicts.

Finally, the solutions given in this chapter are compared, and their pertinence is evaluated in the conclusion.

4.1 Max-Log-MAP scheduling adapted to parallel decoding

Our first direction to increase activity is based on a modified schedule named *SW-*Σ^* that can be used systematically for every turbo code. This new schedule adapted to parallel processing is obtained from schedule *SW-*Σ^0 by sharing some computation units. After a description of the new schedule, we give an expression of its activity, and compare it to the activities of the other schedules *SW-*Σ^- and *SW-*Σ^0.

4.1.1 Introduction of the new schedule

For parallel decoding, two schedules have been considered so far in: *SW-*Σ^- and *SW-*Σ^0 analyzed in chapter 2.2 (see also Appendix A). Their activities are expressed as:

$$\alpha_\Sigma = \beta_\Sigma \cdot \frac{N/P}{N/P + \theta_\Sigma + l}, \qquad \textbf{(4-1)}$$

where $(\theta_\Sigma, \beta_\Sigma)$ = (0, 0.75) for schedule *SW-*Σ^0 and $(\theta_\Sigma, \beta_\Sigma)$ = (L, 1) for schedule *SW-*Σ^-, respectively. We have shown that for a single processor, schedule *SW-*Σ^- uses the processing units more efficiently than its counterpart *SW-*Σ^0. However, with multiple processors, ther former suffers from the constant overhead $\theta_\Sigma + l$, which reduces its activity for a high number of processors and/or small frames. On the other hand, the latter, is less sensitive to the number of processors and to the size of the frame, but its complexity is higher.

For these reasons, with multiple processors a new schedule called Σ^* (and *SW-*Σ^*) is introduced. It combines the advantages of the two preceding schedules: efficient usage of computational resources and lower sensitivity to the number of processors and to the frame size. In addition, the error correcting performance of this schedule is identical to the two others. This schedule and the corresponding architecture are depicted in Figure 4-1. It is built as the superposition of two *SW-*Σ^0 schedules that do not process their respective trellis sections synchronously. The second processor actually starts its decoding after $L/2$ cycles, *i.e.* once the first processor has processed one half of the window. This de-synchronization of the two processors makes it possible to share several computation units between them. In fact, during the second phase of the schedule *SW-*Σ^0 (second half of each window), two soft output computation units are required for each processor: one produces the soft outputs along with the forward recursions, while the other is used with the backward recursions. The duplication of these units, used only half of the time (they are used only half of the time), is responsible for the low activity of the schedule *SW-*Σ^0. Now, for schedule *SW–*Σ^*, these units are used alternatively by the two processors: during cycles $L/2$ to L they produce soft outputs for the first processor, then during cycles L to $3L/2$ they produce soft outputs for the second

processor, and so on. Likewise, the branch metrics computation units are also shared among the two processors: during the first $L/2$ cycles they compute the branch metrics for the forward and backward recursion of the first processor, then during the next $L/2$ cycles they compute the branch metrics for the second processor.

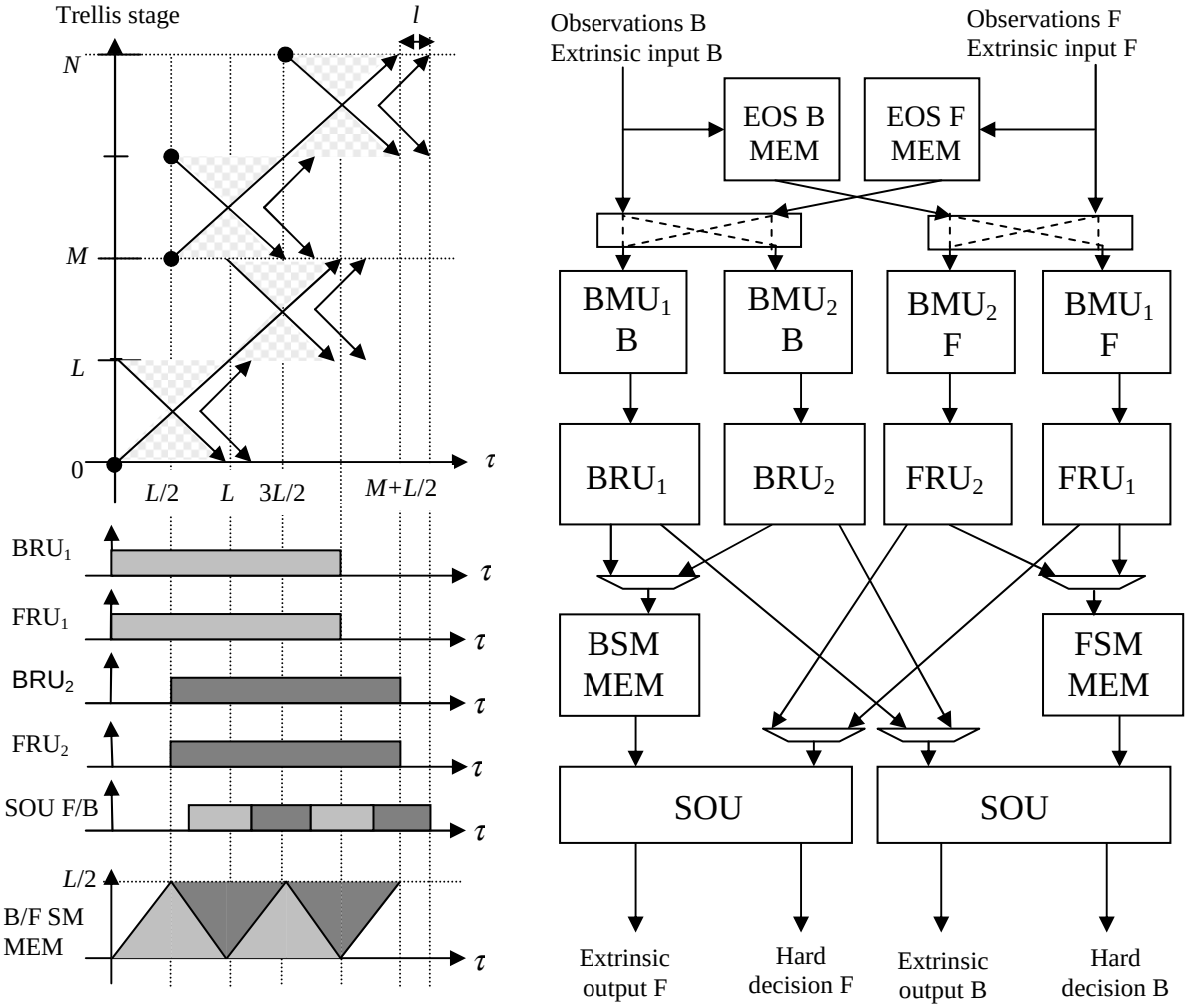


Figure 4-1: Parallel schedule *SW-Σ with *P* = 2 processors and corresponding architecture**

Because of the shared resources, the two processors are designed jointly. Each processor comprises a forward and a backward recursion unit. Two branch metrics computation units and two soft outputs computation units are implemented. Since these resources are shared among the two processors, multiplexers are required to exchange the data between the computation units[1].

4.1.2 Activity of schedule *SW-Σ**

The computational complexity of the architecture performing schedule *SW-Σ** corresponds to the complexity of 4 recursion units, 2 soft output computation units and 4 branch metrics computation units. It is expressed in number of elementary operators as

[1] The suppression these multiplexers, which have a non-negligible impact on the overall complexity, can be achieved easily by modifying the schedule, but it is not described here.

$\Gamma_{\Sigma^*} = 2\left(2\cdot\chi_{SM} + \chi_{SO} + 2\cdot\chi_{BM}\right)$. Therefore, during each cycle, a maximum of Γ_{Σ^*} elementary operations can be performed by the architecture. The schedule requires $N + L/2 + l$ cycles to produce $2.N$ sets of soft outputs. The computational complexity to compute one set of soft outputs for one symbol have been computed in chapter 2.2: $\chi_u = 2\cdot\chi_{SM} + \chi_{SO} + 2\cdot\chi_{BM}$. The activity is thus given by

$$\alpha_{SW-\Sigma^*} = \frac{\chi_u}{\Gamma_{\Sigma^*}}\cdot\frac{M}{M+L/2+l} = \frac{N/P}{N/P+L/2+l}. \qquad \textbf{(4-2)}$$

Like for the schedule *SW-Σ⁻*, the activity tends towards 1 when N tends towards infinity. In other words, the efficiency of the architecture is maximal for long frame sizes. For small frame sizes and or high parallelism, the activity still decreases, but the reduction in activity is caused by the fixed overhead of $L/2 + l$ cycles, which is lower than $L + l$ for *SW-Σ⁻*, however.

4.1.3 Comparison of the activities of the parallel schedules

The activity of the schedule *SW-Σ** can be expressed using the general expression of the activity (2-17) with $(\theta_\Sigma, \beta_\Sigma) = (L/2, 1)$. The activities of the three schedules *SW-Σ⁻*, *SW-Σ** and *SW-Σ⁰* are compared in Figure 4-2 for a window size of $L = 32$. This comparison shows that schedule *SW-Σ** is always[1] preferable to schedule *SW-Σ⁻*. Then, when N/P is lower than a threshold, the activity of the *SW-Σ** schedule is higher than that of schedule *SW-Σ⁰*. Above this threshold, the comparison leads to the opposite conclusion. Compared to *SW-Σ⁻*, this threshold was lowered from $M < 100$ to $M < 20$. Although the schedule *SW-Σ⁰* is less efficient for $M > 20$, its throughput remains the highest.

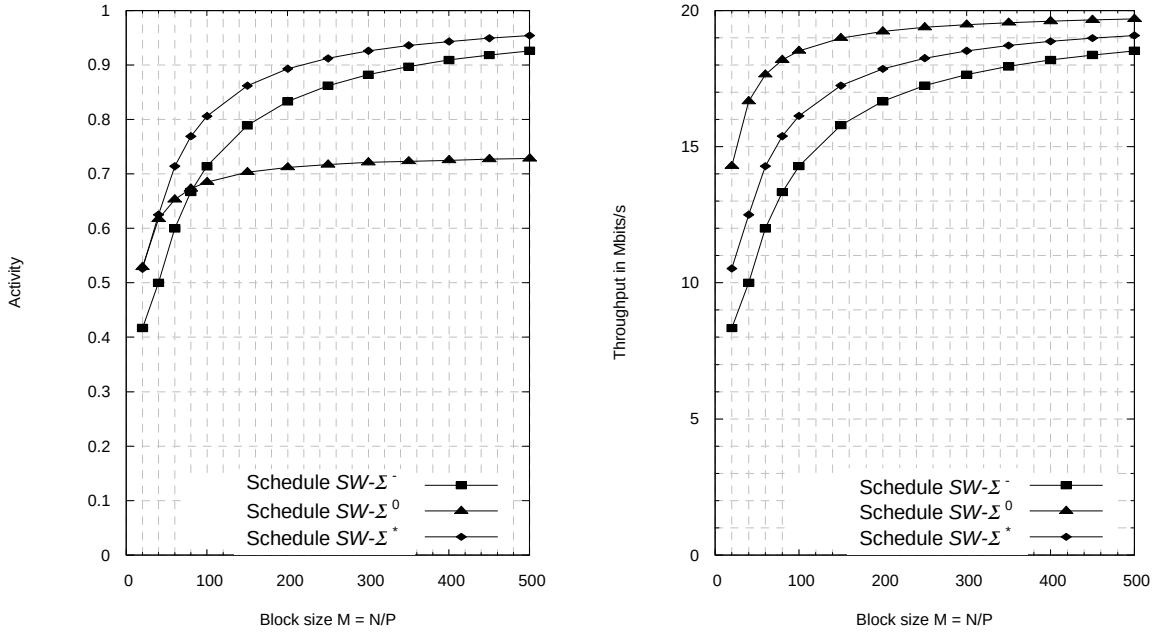


Figure 4-2: Activity and throughput comparison of the schedules *SW-Σ⁻*, *SW-Σ and *SW-Σ⁰* for L=32 as a function of the ratio N/P**

[1] This result stands when schedule *SW-Σ** exists, *i.e.* when $P \geq 2$.

Although the new schedule *SW-Σ** represents a considerable improvement over the other schedules, the activity with small frame size or large number of processors is still penalized. The next two sections will introduce other techniques to overcome this limitation.

4.2 Hybrid parallel architecture

In the second chapter, it has been shown that the conventional method for increasing the throughput is to use a serial architecture, which consists of a cascade of P processors. Then, the throughput of this architecture corresponds to P times the throughput of a single processor architecture. This solution suffers from the duplication of memories, which represents an unaffordable complexity increase, especially for large frame sizes. To overcome this problem, parallel architecture has been introduced. Its key benefits are the reduction in decoding delay and the absence of memory duplication. However, the efficiency of this architecture is limited for small frame sizes: the activity drops, and thus the throughput is reduced accordingly. A hybrid architecture is a partly reconfigurable hardware allowing both serial and parallel decoding. Parallel decoding is thus used to decode large frame sizes, while short frame sizes are decoded via serial processing using the same set of hardware modules.

4.2.1 Description of the hybrid architecture

We assume an architecture with P memory banks of size M and P processors. Using parallel decoding, this architecture can decode frame sizes up to $N_{max} = M \cdot P$. For small frame sizes, a derivation of the serial architecture is used to decode P consecutive frames simultaneously. Instead of implementing a cascade of P component decoders performing a fixed number of iterations, each processor is allocated to the decoding of one frame. In this architecture the degree of parallelism (for each frame) is equal to 1, although a higher degree of parallelism exists at the frame level. The throughput of the decoder corresponds to P times the throughput of a single processor. This variant is much more flexible than the serial architecture since the P processors work independently, handling different frame size with possibly a different number of iterations. For serial decoding, the extrinsic data and channel observations corresponding to P frames need to be stored in the memory banks. This "memory duplication" is performed at no cost if the size of the frame is lower than N_{max} / P. In this case, one memory bank can store the whole set of data relative to one small frame size ($N \leq N_{max} / P$).

The hybrid architecture is depicted in Figure 4-3. An additional observation buffer is required by the parallel architecture, in order to receive the next input frame while decoding the current one. For a degree of parallelism of 1, the received frames are handled as soon as a processor has completed the decoding of its frame and is available to decode a new one. The output memory is also represented. It is divided into P memory banks. The same holds for the extrinsic memory, which is not represented here for the sake of clarity. The programmable network links the memory banks to the processors and has a different configuration according to the degree of parallelism.

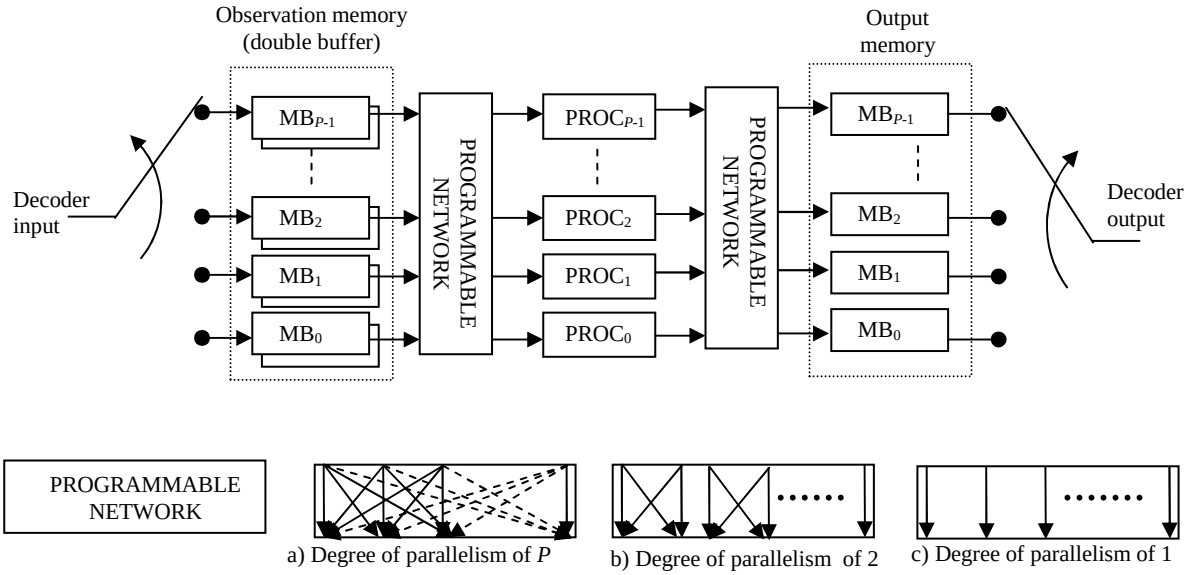


Figure 4-3: Description of the hybrid architecture: a) full parallel architecture with a degree of parallelism of *P*; b) degree of parallelism of 2; c) degree of parallelism of 1 for small frame sizes

The drawback of the hybrid architecture is an increase of the decoding delay for a degree of parallelism of 1. In fact, in this case, it equals the decoding delay d_S of a serial architecture. For schedule *SW-Σ*$^-$, it is expressed as

$$d_S = \frac{1}{f_{clk}} \cdot 2 \cdot I_{max}(N + L + l). \qquad \textbf{(4-3)}$$

The implementation of an architecture with a parallelism of 1 is not straightforward for the schedule *SW-Σ**. Actually, because of the resource sharing between two processors, this latter cannot work on two frames completely independently. We have derived a solution to this problem, but it is not described in this manuscript.

4.2.2 Generalization of the hybrid architecture

The hybrid architecture can be generalized to other degrees of parallelism. Let us consider that P is a power of two. The full parallel architecture corresponds to a degree of parallelism $d_P = P$, whereas the serial architecture decodes with a degree $d_P = 1$. Several other degrees of parallelism between $d_P = P$ and $d_P = 1$ can be devised. In particular, the degrees $P/2$, $P/4$, etc. can be considered. A degree of parallelism $d_P = 2$ is used in Figure 4-3.b. The decoding delays associated with the different configurations of the hybrid architecture are therefore expressed as a function of the degree of parallelism as

$$d_{TD}(d_p) = \frac{1}{f_{clk}} \cdot 2 \cdot I_{max}(N / d_p + \theta_\Sigma + l). \qquad \textbf{(4-4)}$$

Likewise, the activity of the hybrid architecture depends on the current degree of parallelism d_P:

$$\alpha_{SW-\Sigma}(d_p) = \beta_\Sigma \cdot \frac{N / d_p}{N / d_p + \theta_\Sigma + l}. \qquad \textbf{(4-5)}$$

These relations show that decreasing the degree of parallelism increases the activity of the architecture, and thus its throughput, but at the cost of an increased decoding delay.

The optimization of the hybrid architecture involves taking specified thresholds on the frame sizes in order to switch to a configuration with a higher or a lower degree of parallelism. This concern is discussed in the next section.

4.2.3 Application of the hybrid architecture

The hybrid architecture is used when the decoder needs to support a large range of frame sizes including small, medium and large frame sizes. The optimization of the threshold for switching from a configuration with a degree of parallelism d_P to another configuration is performed according to several parameters:

- the number of processors P, which fixes the maximum number $d_P = P$.
- the maximal frame size N_{max} supported by the decoder, which fixes the size N_{max} / P of the memory banks.
- the maximal decoding delay of the decoder, specified by the requirement of the application.
- the minimal throughput of the decoder.

Example 4-1:

An example of an application of the hybrid architecture is represented in Figure 4-4 for the 802.16 standard [WiMAX]. The hybrid architecture is compared to a serial only architecture and a parallel only architecture. The frame size of this double-binary turbo code ranges from $N = 24$ to 2400 double-binary symbols. We use an architecture with $P = 8$ processors and all the available degrees of parallelism in order to maintain a given throughput. The throughput is here normalized relatively to the throughput that an architecture with maximal activity would reach after 8 iterations. The decoding delay is normalized relatively to the transmission delay assuming a QPSK modulation and a rate 1/2.

In order to maintain a normalized throughput of 1, the number of iterations I_{max} is decreased accordingly. Then, the architecture switches between the different configurations: $d_P = 8$ for $1200 < N \leq 2400$, $d_P = 4$ for $600 < N \leq 1200$, $d_P = 2$ for $600 < N \leq 1200$, $d_P = 1$ for $300 < N \leq 600$. The curves presented in Figure 4-4 clearly show that the hybrid architectures offers the advantages of both serial and parallel decoding at the two ends of the frame size range. This solution enables a constant number of iterations, except for the smallest frame sizes, for which the number of iterations needs to be reduced significantly. The drawback of this solution lies in the decoding delay. The decoding delay is bounded by the delay for the longest frame size. However, it becomes much more important than the transmission delay when d_p is decreased.

This example provides one application of the hybrid architecture. However, we have not taken into account the influence on the throughput and latency of switching from one

configuration to another one. To do that, an efficient design of a hybrid architecture would take into account the high-level constraints and the characteristics of the application, which can be provided by the system designer: probability distribution of the frame size, time distribution of the frame size, etc. This work has not been done so far, but is one of our interesting perspectives.

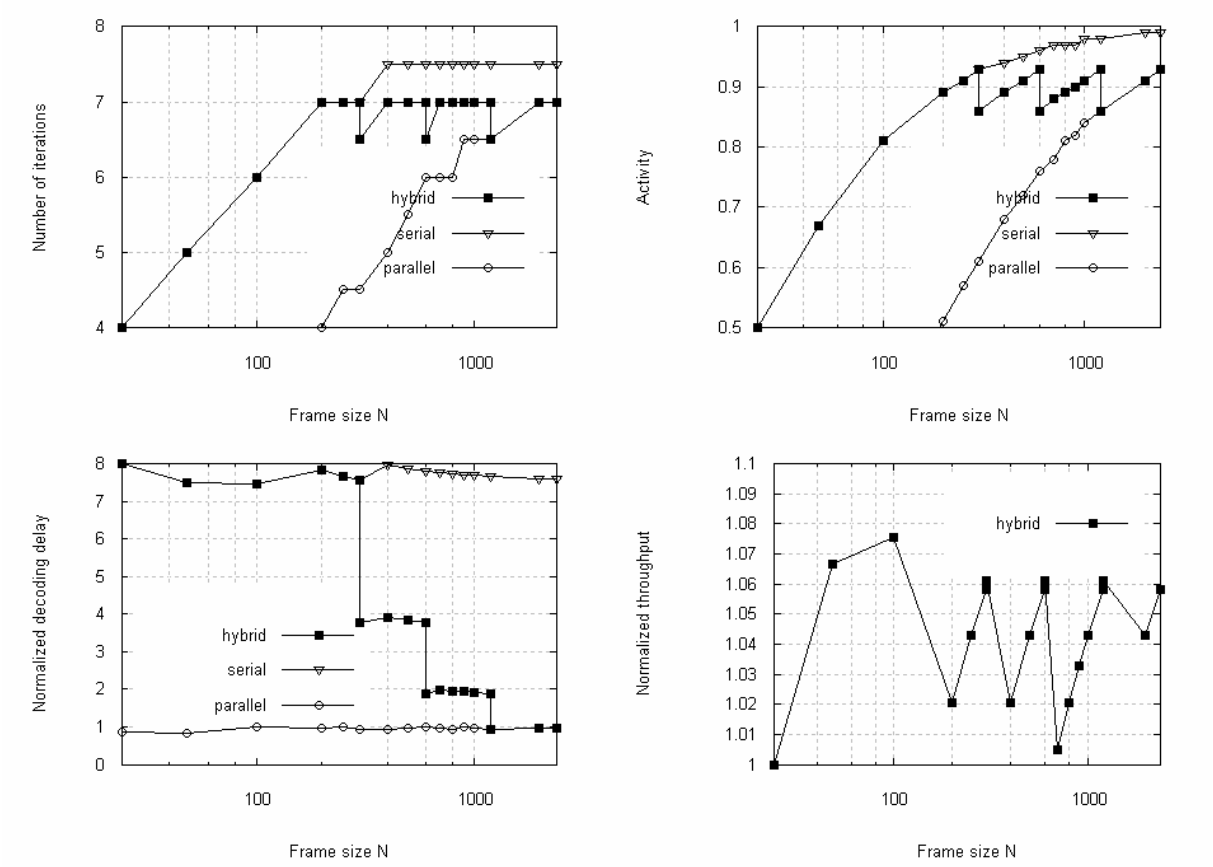


Figure 4-4: Example of utilization of the hybrid architecture for the WiMAX standard

4.3 Overlapping schedule for increased activity

In chapter 2, we mentioned a solution yielding a significant improvement in activity in the context of iterative decoding. This solution relies on an overlapping when processing consecutive half-iterations. In this section we will show that this may lead to consistency conflicts: the extrinsic input required at the beginning of the current iteration has not yet been produced at the previous iteration.

After a description of the overlapping of consecutive half-iterations and its potential, the definition of a consistency conflict is given. Then, an analytical evaluation of the average number of consistency conflicts for different schedules is performed. The results of this evaluation provide justification for the need to find techniques avoiding consistency conflicts. Such techniques will be discussed in the last part of this section.

4.3.1 Overlapping of consecutive half-iterations

In this section we introduce the concept of overlapping in the processing of consecutive half-iterations and we also discuss the associated activity augmentation.

Let us first describe non-overlapping processing in Figure 4-5, where the processing of the next half-iteration is started once the previous is totally completed. For a given schedule Σ, the processing of a half-iteration is completed once the pipeline has been emptied, *i.e.* when the last extrinsic output has been written into the extrinsic memory. Due to the pipeline depth, l cycles are required in addition to the $M+\theta_\Sigma$ cycles required by the execution of the schedule.

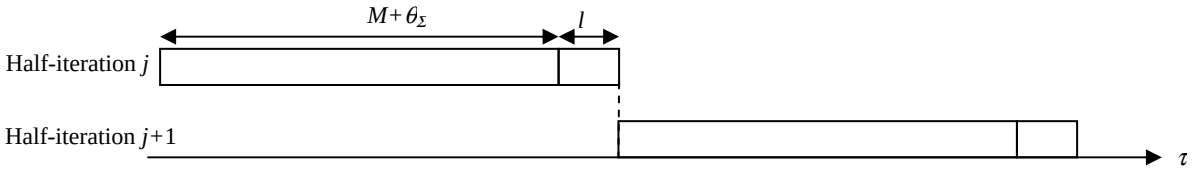


Figure 4-5: No overlapping in the processing of consecutive half-iterations

The overlapping of consecutive half-iterations is achieved by starting the next half-iteration before the end of the current half-iteration, as depicted in Figure 4-6. The depth of the overlapping region expressed in number of clock cycles is denoted Δ. It is bounded by $\Delta_{max} = \theta_\Sigma + l$, whose value depends on the availability of the computation resources of the schedule considered: $\Delta_{max} = L+l$ for schedule $SW\text{-}\Sigma^-$, $\Delta_{max} = L/2+l$ for schedule $SW\text{-}\Sigma^*$ and $\Delta_{max} = l$ for schedule $SW\text{-}\Sigma^0$. This maximum overlapping depth shows that the fixed overhead $\theta_\Sigma + l$ lost when switching from the current half-iteration to the next one can be recovered. The most important activity gain is obtained for the schedule with the highest fixed overhead, *i.e.* schedule $SW\text{-}\Sigma^-$. The sliding window schedules with overlapping depth Δ are denoted $OSW\text{-}\Sigma^-(\Delta)$, $OSW\text{-}\Sigma^0(\Delta)$ and $OSW\text{-}\Sigma^*(\Delta)$.

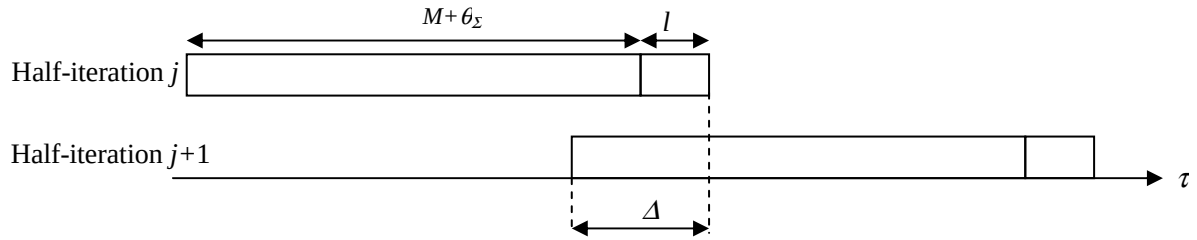


Figure 4-6: Overlapping in the processing of consecutive half-iterations

A general expression of the activity as a function of the overlapping depth Δ and the maximum number of iterations I_{max} for a given schedule Σ is given by:

$$\alpha_\Sigma = \beta_\Sigma \cdot \frac{I_{max} \cdot (N/P)}{I_{max} \cdot (N/P + \theta_\Sigma + l - \Delta)}. \qquad \textbf{(4-6)}$$

We note that for the last half-iteration, no consistency conflicts can occur since the decoding of the first half-iteration of the next frame is independent of the current frame. Let Δ_j denote the overlapping depth for half-iteration j, $\Delta_j = \Delta_{max} = \theta_\Sigma + l$ for the last half-iteration, $\Delta_j = \Delta$ otherwise. The activity is then modified to:

$$\alpha_{\Sigma} = \beta_{\Sigma} \cdot \frac{2 \cdot I_{max} \cdot (N/P)}{2 \cdot I_{max} \cdot (N/P + \theta_{\Sigma} + l) - \sum_{j=0}^{2 \cdot Imax-1} \Delta_j} . \quad \textbf{(4-7)}$$

This last relation shows that in the context of iterative decoding, using a maximal overlapping depth Δ_{max} enables the bound of the activity β_{Σ} achieved for infinite frame sizes to be reached. Unfortunately, this solution leads to another type of memory conflict, which we term a consistency conflict: due to interleaving/de-interleaving the extrinsic input required at the beginning of the next half-iteration might not yet have been produced by the current half-iteration. The next section describes the appearance of this phenomenon.

4.3.2 Consistency conflicts

Figure 4-7 illustrates the appearance of a consistency conflict. Let us assume that the current half-iteration j produces extrinsic output vector $\mathbf{Z}^{(j)}$. Assuming that the extrinsic information is stored in a single memory, vector $\mathbf{Z}^{(j)}$ updates the previous extrinsic vector $\mathbf{Z}^{(j-1)}$ stored in the memory. A consistency conflict occurs when the next half-iteration $j + 1$ requires the extrinsic value $\mathbf{Z}_k^{(j)}$ at time τ for trellis index k' ($k = k'$ when decoding in the natural order, $k = \Pi(k')$ otherwise) and when this value has not yet been produced by the current iteration j. Consequently, instead of $\mathbf{Z}_k^{(j)}$, the extrinsic input read during the next half-iteration corresponds to the value stored in the memory, *i.e.* the extrinsic output produced by the same component decoder at the previous iteration $\mathbf{Z}_k^{(j-1)}$. This violates the iterative decoding principle since the same information produced by one decoder is fed back to itself, resulting in significant performance degradation.

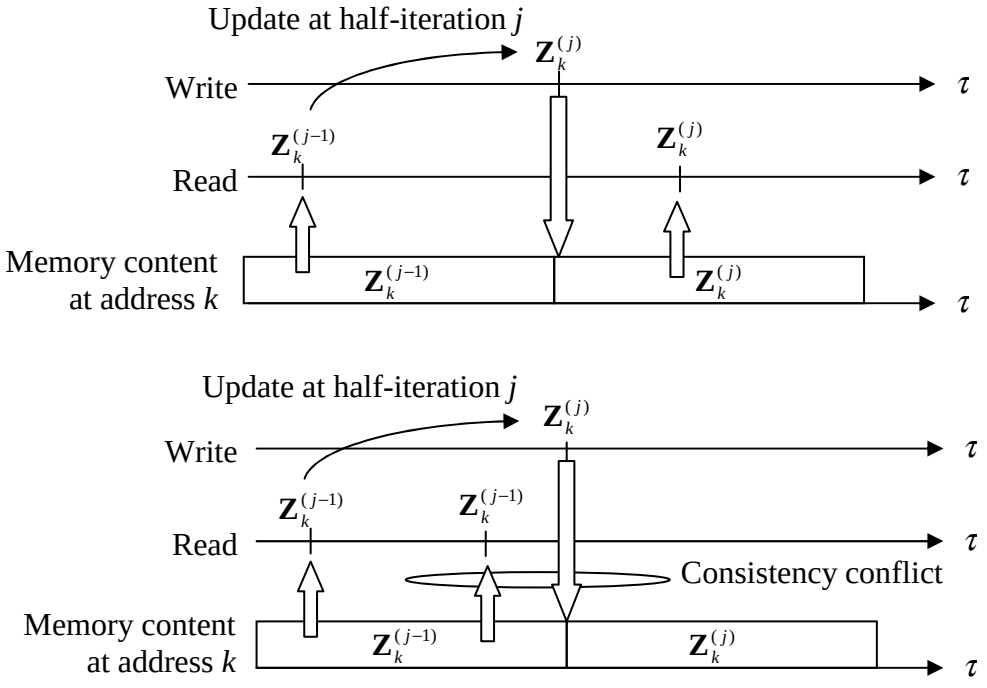


Figure 4-7: Absence of a consistency conflict / Presence of a consistency conflict

The next section evaluates the average number of consistency conflicts as a function of the overlapping depth Δ.

4.3.3 Evaluation of the average number of consistency conflicts

In this sub-section, we develop an analytical expression of the average number of consistency conflicts ψ. To this end, we introduce the concept of uniform random interleaver that will be used to upper bound the number of consistency conflicts for uniform interleavers. Then, we will develop the expression of ψ for a single processor, before generalizing for multi-processor architectures and specific schedules. Finally, our model will be validated by listing the effective number of consistency conflicts for existing multi-processors implementations.

4.3.3.1 Uniform interleaver

Let us introduce the notion of uniform interleaver as follows.

Definition 4-1: A uniform random interleaver of length N maps each symbol of the natural order to a given symbol of the interleaved order with probability $1/N$.

Consequently, one symbol of the natural order is interleaved to a region of length u symbols with probability u / M. In other words, uniform random interleavers spread the symbols equally over the whole frame. It is worth noting that the interleavers that are optimized for performance of the code tends to verify the property of uniform interleaving as N increases. In fact, for good error decoding performance, the symbols that are close to each other in one dimension are spread throughout the frame in the other dimension. This spreading of the symbols improves the convergence of the code and also its minimum distance. For example, ARP interleavers and hierarchical interleavers for MSTCs developed in the previous chapter verify the property of uniform interleaving.

4.3.3.2 Upper bound for a uniform interleaver

Using the concept of uniform random interleaver, an upper bound for uniform interleavers of the average number of consistency conflicts ψ can be expressed. Let us consider that the two half-iterations overlap on Δ symbols. Two different overlapping regions are considered, depending on half-iteration currently processed. The first region $\Omega_{n\rightarrow i}$ corresponds to the transition $T_{n\rightarrow i}$ of processing in the natural order to processing in the interleaved order. The second one $\Omega_{i\rightarrow n}$ corresponds to the transition $T_{i\rightarrow n}$ of the interleaved to the natural order. In the first step, ψ is calculated for a single overlapping region.

For each of the first Δ symbols read during the beginning of the current half-iteration, a consistency conflict can occur only if it corresponds to a symbol produced during the last Δ cycles of the previous half-iteration (see Figure 4-8). Since each symbol is mapped with a probability Δ / N to the symbols produced during the last Δ cycles, Δ^2 / N symbols among the first Δ symbols are mapped to symbols of the overlapping region. The upper bound on ψ for an overlapping depth of Δ symbols is thus given by

$$\psi_{\max} = \frac{\Delta^2}{N}. \qquad \textbf{(4-8)}$$

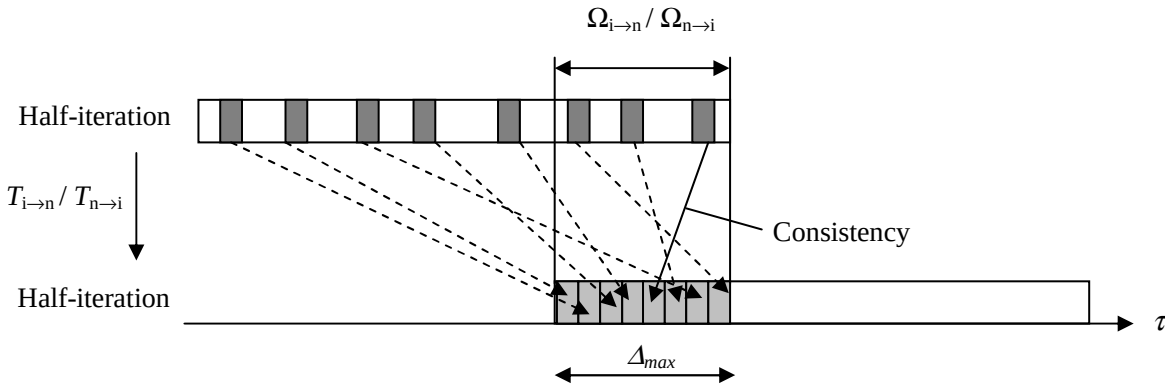


Figure 4-8: Uniform interleaving of symbols of the overlapping region

4.3.3.3 Average number of consistency conflicts for a single processor

Let us now take into account the uniform interleaver property to derive the expression of ψ for a single processor.

Not all Δ^2 / N symbols from the overlapping region read at the beginning of the half-iteration that are mapped in the other dimension to the overlapping region necessarily correspond to a consistency conflict. Indeed, the probability for a symbol to generate a consistency conflict is not constant for all symbols of the overlapping region. The first symbol that is read during the beginning of the half-iteration has a much higher probability to be associated with a consistency conflict than the symbol that is read at the end of this region. For instance, no consistency conflict occurs for the symbol that is read at time $\tau = \Delta / 2$, if this symbol is mapped to a position corresponding to the first half of the overlapping region (see Figure 4-9.a). If the symbol is mapped to the second half, a consistency conflict occurs (see Figure 4-9.b). Hence, the probability for this symbol to produce a consistency conflict is $\Delta / (2N)$. For symbols before and beyond this threshold, the probabilities are higher and lower, respectively.

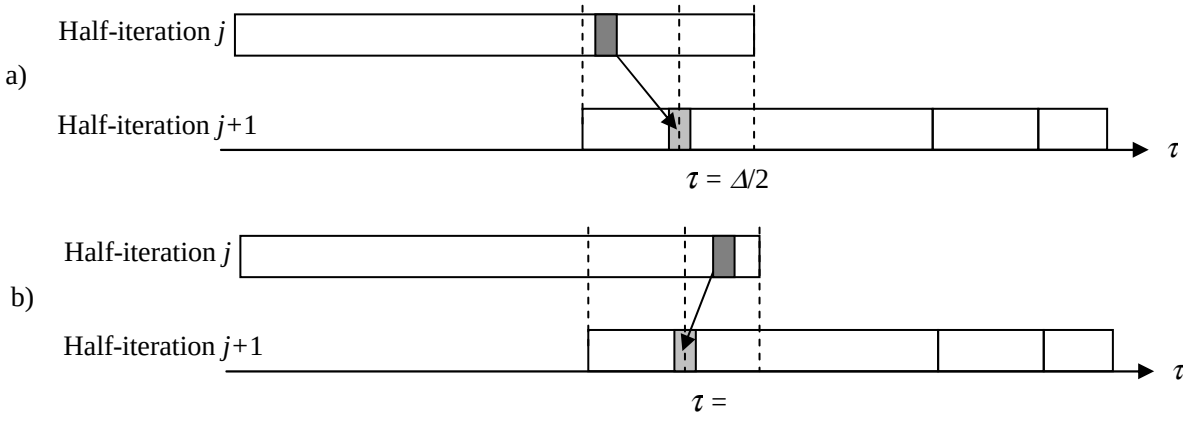


Figure 4-9: Symbols of the overlapping region mapped to the overlapping region in the other dimension: a) a consistency conflict is associated with the symbol; b) no consistency conflict is associated with the symbol

Under the assumption of uniform random interleaving, the probability for a variable read at time τ to be concerned by a consistency conflict equals

$$p_\tau = \frac{\Delta - \tau}{N}. \quad \textbf{(4-9)}$$

A consistency conflicts occurs if the symbol is mapped to a symbol generated during the last Δ - τ cycles of the previous half-iteration. The average number of consistency conflicts ψ is thus obtained by considering all symbols read from time τ = 0 to Δ-1:

$$\psi = \sum_{\tau=0}^{\Delta-1} p_\tau = \sum_{\tau=0}^{\Delta-1} \frac{\Delta - \tau}{N}. \quad \textbf{(4-10)}$$

Using the change of variable $\tau' = \Delta - \tau$ in (4-10) yields

$$\psi = \frac{1}{N} \sum_{\tau'=1}^{\Delta} \tau', \quad \textbf{(4-11)}$$

where the sum corresponds to the sum of the Δ first integers. Thus ψ is given by

$$\psi = \frac{\Delta(\Delta+1)}{2N}, \quad \textbf{(4-12)}$$

which we approximate to

$$\psi = \frac{\Delta^2}{2N}. \quad \textbf{(4-13)}$$

This approximation can be found easily by considering that the overlapping region includes Δ symbols interleaved to a region of Δ symbols with probability $\Delta / (N)$. The coefficient 1/2 is used to take into account the fact that around half of these symbols are associated with consistency conflicts.

The total number of conflicts for an interleaver corresponds to the addition of the number of conflicts for the two overlapping regions $\Omega_{i\to n}$ and $\Omega_{n\to i}$, *i.e.* around twice the number for a single region. Let us also define the average proportion of consistency conflicts $\eta = \psi / N$ that will be used later on.

4.3.3.4 Generalization to multiple processors.

In this section, we generalize the expression of ψ to multi-processor architectures implementing schedules *OSW-*$\Sigma^-(\Delta)$, *OSW-*$\Sigma^0(\Delta)$ and *OSW-*$\Sigma^*(\Delta)$, where $\Delta \le \Delta_{max}$.

Schedule *OSW-*$\Sigma^-(\Delta)$, Δ_{max} = L + *l*:

For *P* processors implementing schedule *OSW-*$\Sigma^-(\Delta)$, *P* read accesses and *P* write accesses are performed at each cycle to the extrinsic memory. The overlapping region therefore covers $P \cdot \Delta$ symbols that are associated with consistency conflicts if they are interleaved to the counterpart region of $P \cdot \Delta$ symbols. ψ is then expressed as

$$\psi_{SW-\Sigma^-} = \frac{\Delta^2 P^2}{2N}. \quad \textbf{(4-14)}$$

Schedule $OSW\text{-}\Sigma^0(\Delta)$, $\Delta_{max} = l$:

Each processor performs two read accesses and two write accesses at each cycle. Therefore, the overlapping region covers $2P\cdot\Delta$ symbols and the average number of consistency conflicts is multiplied by 4 compared to schedule $SW\text{-}\Sigma^-$:

$$\psi_{SW-\Sigma^0} = 2\frac{\Delta^2 P^2}{N}. \quad \textbf{(4-15)}$$

Schedule $OSW\text{-}\Sigma^*(\Delta)$, $\Delta_{max} = L/2 + l$:

The overlapping region covers a total of $P\cdot\Delta$ symbols, since during each cycle, each processor performs either two read or two write accesses to the extrinsic memory: half of the processors perform read operations, while the other half perform write operations. ψ is therefore equal to that of schedule $SW\text{-}\Sigma^-$:

$$\psi_{SW-\Sigma^*} = \psi_{SW-\Sigma^-} = \frac{\Delta^2 P^2}{2N}. \quad \textbf{(4-16)}$$

From these expressions, we observe that for the three schedules ψ depends quadratically on P and on Δ. In other words, a division by 2 of Δ results in a division by 4 of ψ. This observation leads to the following additional advantage of schedule $OSW\text{-}\Sigma^*$ over schedule $OSW\text{-}\Sigma^-$ described as follows. Expressions (4-14) and (4-16) indicate that for the same overlapping depth, both schedules have on average the same number of consistency conflicts. However, a maximal activity of 1 is not obtained for the same maximal overlapping depth: $\Delta_{max} = L+l$ for $OSW\text{-}\Sigma^-$ and $\Delta_{max} = L/2+l$ for $OSW\text{-}\Sigma^*$. Hence, for schedule $OSW\text{-}\Sigma^*$ Δ_{max} is almost half the value associated with schedule $OSW\text{-}\Sigma^-$, and the number of conflicts is almost four times lower. The lower number of consistency conflicts will have a positive influence for facilitating their handling as presented in sections 4.4 and 4.5.

4.3.3.5 Validation of the model

In order to validate the model of the average number of consistency conflicts, we have computed the effective number of consistency conflicts for several deterministic ARP interleavers. The resulting numbers are represented by dots in Figure 4-10, which represents the proportion of consistency conflicts $\eta = \psi / N$ as a function of the size of the block $M = N/P$ ($M \geq L$) for a maximal overlapping depth $\Delta_{max} = L + l$ for schedule $OSW\text{-}\Sigma^-$. The same comparison conducted for schedule $OSW\text{-}\Sigma^*$ is also represented.

Figure 4-11 compares the evolution of the number of consistency conflicts for $P = 8$ processors as a function of the overlapping depth Δ. It also shows the improvement in activity obtained for schedules $OSW\text{-}\Sigma^-$, $OSW\text{-}\Sigma^*$ and $OSW\text{-}\Sigma^0$. For the three schedules, the activity is increased to 1 when the overlapping depth augments. A maximal activity of 1 is reached for schedule $OSW\text{-}\Sigma^*$ and $OSW\text{-}\Sigma^-$ with an overlapping depth of 24 and 40, respectively, corresponding to a proportion of consistency conflicts of 0.017 and 0.046, respectively. For schedule $OSW\text{-}\Sigma^0$, the maximal activity is upper-bounded by $\beta_\Sigma = 0.74$.

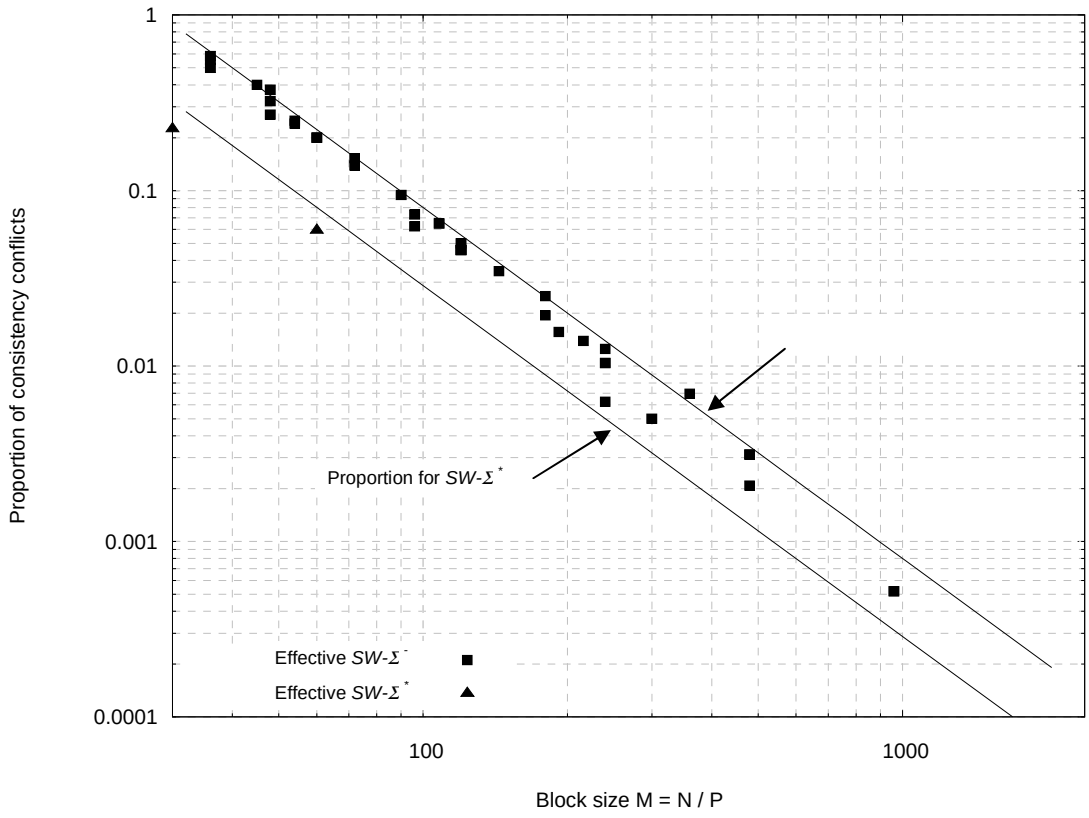


Figure 4-10: Proportion of consistency conflicts as a function of M for $\Delta = \Delta_{max}$, $L = 32$, $l = 8$

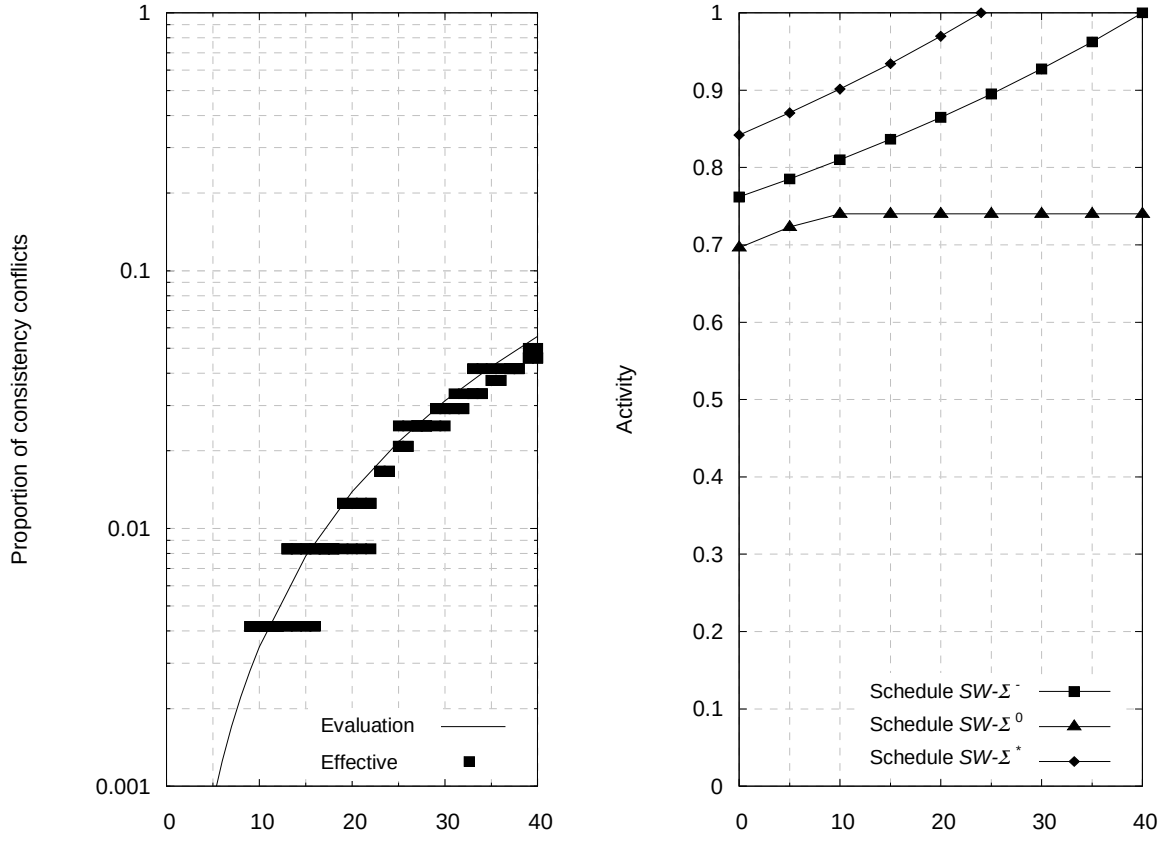


Figure 4-11: Proportion of consistency conflicts as a function of Δ for N/P =120

These comparisons demonstrate the validity of our model for evaluating the average number of consistency conflicts for a given interleaver. However, no solution has yet been proposed to handle these conflicts.

4.3.4 Solutions for resolving consistency conflicts

Like for the parallel conflicts of chapter 3, three different techniques can be considered to resolve the problem of consistency conflicts, differing by the stage of the design process at which they take place:

- execution-stage solutions: additional ad-hoc hardware is used to resolve the consistency conflicts. However, unlike the solution for the parallel conflicts, this solution has a negative impact on performance. This solution is one of our original contributions and will be addressed in section 1.4.
- compilation-stage solutions: this involves the selection of an appropriate schedule in order to prevent or at least reduce the presence of consistency conflicts. In particular, the aim is to read another extrinsic input instead of the conflicting one. But, due to the sequentially forward and backward recursions, this solution cannot be applied. For this reason, the schedule should work on a window per window basis: the windows are decoded using an order such that it uses only the extrinsic data already produced. The schedule is no longer a "sliding" window schedule, but a "leaping" window schedule where the recursions are initialized using the NII technique (described in chapter 1.5). But in order to be able to suppress all the consistency conflicts, the size of the windows needs to be reduced drastically. We believe this would lead to a severe degradation in the error decoding performance, thus spoiling the activity improvement. Note however that we have applied this solution successfully to the design of a high-speed parallel LDPC decoder [Tousch *et al*.-04].
- design-stage solutions: the interleaver and the architecture (and also the decoding schedule) are designed jointly in order to prevent the appearance of any consistency conflicts. This constrained interleaver design is addressed in section 1.5. It constitutes a major contribution of this thesis.

Let us start with the solution handling conflicts during the execution of the algorithm, which requires no modification of the code itself.

4.4 Execution-stage solution

This section addresses an ad-hoc architectural solution that, during the execution of the algorithm, solves the consistency conflicts associated with the overlapping of consecutive half-iteration described in the previous section. The idea is based on the utilization of extrinsic information from the previous iteration. After introducing this technique, we discuss the architectural solutions that can be employed to implement it. Finally, we analyse the impact of the performance of this solution as a function of the overlapping depth.

4.4.1 Overview

As shown in section 4.3.2, the utilization of our conventional parallel architecture with a single extrinsic memory violates the principle of iterative decoding if no special care is taken of the consistency conflicts. In fact, let us consider that the current half-iteration requires the extrinsic data $\mathbf{Z}_k^{(j-1)}$. It has not yet been written into the memory at the previous iteration, so the corresponding memory address still contains the extrinsic data $\mathbf{Z}_k^{(j-2)}$ produced by the same component decoder at the previous iteration. This feedback of the information generated by one decoder to the same decoder drastically degrades the performance of the decoder.

In order to avoid this feedback, the extrinsic data that has not yet been produced is replaced by the extrinsic data of the same component decoder produced at the previous iteration. In other words, at the current iteration *j*, if the extrinsic data $\mathbf{Z}_k^{(j-1)}$ is not yet available, then the extrinsic data $\mathbf{Z}_k^{(j-3)}$, which can be seen as an "old" version of $\mathbf{Z}_k^{(j-1)}$, is used instead (see Figure 4-12).

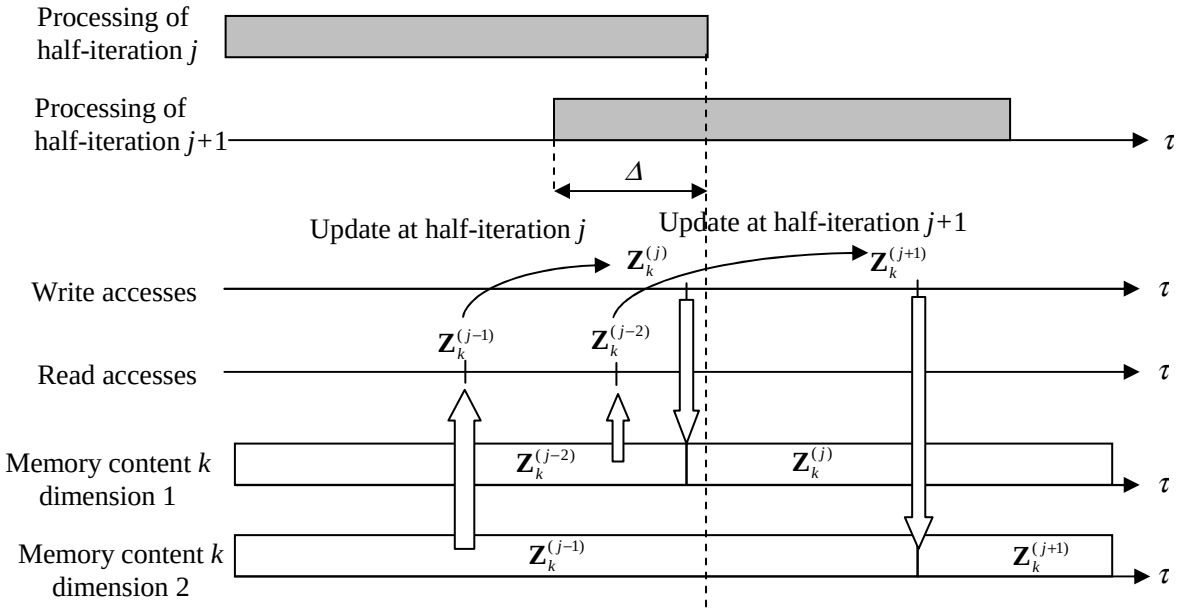


Figure 4-12: Utilization of extrinsic data generated at the previous iteration to resolve a consistency conflict

Figure 4-13 illustrates the decoding scheme, with a proportion η of consistency conflicts, for which the extrinsic data of the previous iteration is used. The extreme case (η =1), when all symbols lead to a consistency conflict and all extrinsic inputs correspond to previous-iteration data, exhibits three independent processes that do not share information together. Obviously, the performance in this case corresponds to the performance obtained by a division by three of the number of iterations. The degradation of this scheme will be addressed after the description of a hardware architecture to handle the consistency conflicts.

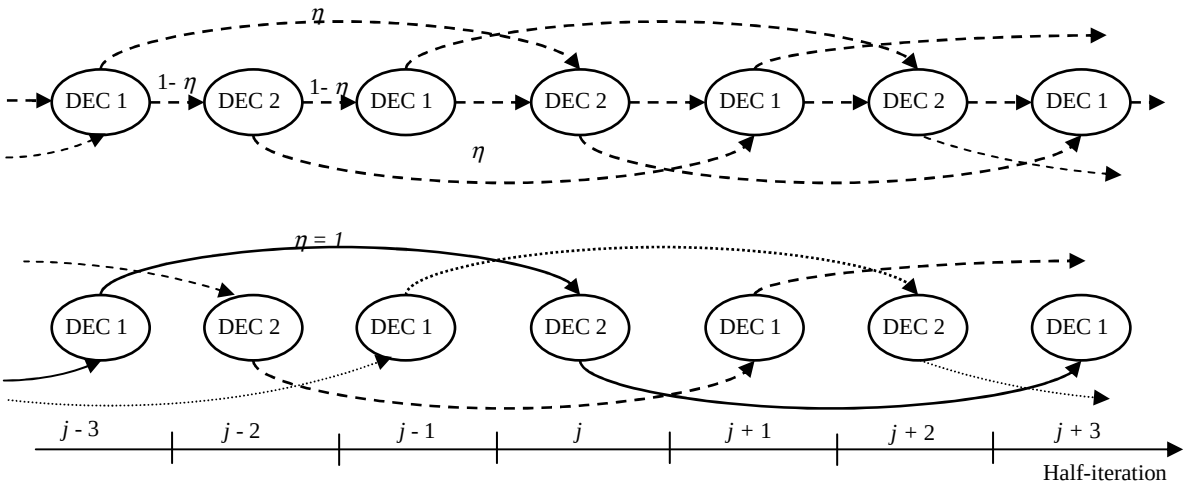


Figure 4-13: Decoding scheme utilizing extrinsic information from the previous iteration

4.4.2 Architecture

The solution described above requires storing extrinsic data of the previous iteration. In order to implement this solution, we derive three different architectures. The first one duplicates the extrinsic memory. Then, we introduce a second architecture with minimal memory requirement, but with complex address computations. Finally, we propose our architecture, which trades off the memory requirement and the complexity of the address computation.

4.4.2.1 Duplication of the extrinsic memory

This architecture simply duplicates the extrinsic memories: one memory E_n for storing extrinsic information in the natural order, and another E_i for the interleaved order. When decoding in the natural order, the processor reads from E_i and writes its extrinsic outputs to E_n, and vice versa for decoding in the interleaved order. Then, when extrinsic input $\mathbf{Z}^{(j-1)}$ is required at half-iteration j and it has not yet been updated, the value read from the memory corresponds to $\mathbf{Z}^{(j-3)}$. For this architecture, the memory requirement is twice the extrinsic memory required for a conventional iterative architecture. In fact, this technique represents another illustration of the first architectural law (see chapter 2.1): the gain in computational power is paid for by an increase in memory. However, in this case it also comes at the cost of performance degradation (see section 4.4.3).

The duplication of the memory is not necessary. In fact, only the symbols that belong to the overlapping regions $\Omega_{i\to n}$ and $\Omega_{n\to i}$ might be associated with consistency conflicts. This property is used to explore other architectures in the two next sections.

4.4.2.2 Architecture with two minimal secondary extrinsic memories

For an overlapping region of Δ cycles and P processors, the number of symbols in the overlapping region is at maximum $P{\cdot}\Delta$ for schedules $OSW\text{-}\Sigma^{-}$ and $OSW\text{-}\Sigma^{*}$. Consequently, for the two overlapping regions, two memories of size $P{\cdot}\Delta$ words are sufficient. The

architecture therefore comprises a principal extrinsic memory M_P for storing the N extrinsic data that are the most up to date and two secondary memories denoted M_{S_1} and M_{S_2} of size $P \cdot \Delta$ for the natural and interleaved order, respectively. However, a memory organization for storing the data in the secondary memories is required, specifying the address at which each extrinsic variable is stored.

Let us first consider the case of a single processor performing the *OSW*-Σ^- schedule. A first solution involves storing the data in the "production" order with a simple incremental addressing: the first variable of the overlapping region is stored at address 0, while the last one is stored at address Δ-1. In the other dimension, due to interleaving/de-interleaving, these memories are not read in the same order. And, computing the address at which the corresponding variable is stored requires additional circuitry, which might be complex.

For instance, let us assume that during the decoding in the natural order, data with indices N-Δ to N-1 of the overlapping region $\Omega_{n \to i}$ are written respectively and successively to address 0 to Δ-1 of the secondary memory. Then, during the processing in the interleaved order, if th extrinsic input for the symbol with index k is read at address $\Pi(k)$ such that $\Delta \leq \Pi(k) < N$, then its address in the secondary memory can be computed easily by $\Pi(k) - (N-\Delta)$. Similarly, for the other overlapping region $\Omega_{i \to n}$, the same procedure holds, but using the inverse permutation Π^{-1} instead of Π. Generally, a turbo decoder architecture implements either the interleaving function or the deinterleaving function by means of a circuitry to generate the (de)interleaved addresses or a memory for storing the latter. The solution described in this section requires implementing the two functions, which represents an important increase in complexity.

In conclusion, in order to store the extrinsic information to resolve the consistency conflicts, we have two possible solutions. On the one hand, the first solution requires no complex address computation, at the cost of a duplication of the entire extrinsic memory. On the other hand, the memory contains only the extrinsic information corresponding to symbols in the overlapping regions, but at the cost of a more complex address computation. A more optimal solution presented now falls between these two extreme solutions: using a hash table.

4.4.2.3 Architecture with a hash-table

This section introduces an architecture associated with two secondary extrinsic memories of reasonable size whose addresses can be computed easily. After a brief description of a hash-table, we use this concept to design our architecture for turbo codes associated with ARP interleavers.

A hash table is a memorization structure where the data are accessed directly, *i.e.* there is a direct mapping between the data and the address at which they are stored in the table. Actually, each data is linked to a key, which is computed from the data thanks to a hash function h. A hash table comprises a table of size Λ, where the data are stored at the address corresponding to their key. A hash table is efficient when the set of data that is stored in the

hash table is bounded and when the key associated with one data sample is unique. Generally, the size of the hash table with unique keys is greater than the number of data it can store.

The concept of hash table can be used to find an optimal solution with a small memory requirement for storing the data in the overlapping region, and a direct address computation to access these data. The data of the secondary memories M_{S_1} and M_{S_2} are stored relatively to their indices in the natural order as is the case for the principal memory M_P. Finding an efficient hash-table relies on the existence of a simple hash function associated with a given interleaver.

Let us consider the case of the ARP interleaver whose equation is given by:

$$\Pi(k) = \lambda \cdot k + \mu(k \bmod 4) \bmod N \quad \textbf{(4-17)}$$

Let $\Lambda = 2^{\lambda}$ be a power of two higher than the overlapping depth Δ, $\Lambda \geq \Delta$, and let us consider the function h whose expression is given for each index k = 0 to N-1 as:

$$h(k) = k \bmod \Lambda \quad \textbf{(4-18)}$$

From this relation, it follows that for Δ different indices k, function h associates Δ different keys. Hence, h can be used as a hash function associated with a memory of Λ words for storing the data relative to the overlapping region $\Omega_{i \to n}$.

An important property of ARP interleavers is that they preserve the classes of congruence in the interleaved order. In fact, all symbols whose indices are equal to e modulo an integer E ($E \leq N$, E multiple of 4) in the natural order are interleaved to indices that are equal to e' modulo E in the interleaved order. Since the Δ indices k corresponds to Δ distinct classes modulo Λ, the interleaved indices $\Pi(k)$ correspond to distinct classes. Hence, h associates $\Pi(k)$ distinct keys for each of the indices, and h can be used as a hash function associated with a memory of Λ words for storing the data relative for $\Omega_{n \to i}$.

In conclusion, h can be used as a hash function for the two secondary memories of Λ words. The corresponding architecture is depicted in Figure 4-14. The addresses of the data are computed in a straightforward way thanks to the computation of a modulo with a power of 2. Nevertheless, another problem needs to be solved with such a hash function. How can we distinguish easily whenever the symbol is associated with a consistency conflict or not? A simple solution is used as follows. This solution adds an additional bit ω_k for each index k of the principal memory, which indicates whether the symbol is associated with a consistency conflict. Then, the algorithm described below can be used. For this algorithm, sub-index $_O$ indicates the processing order (natural or interleaved), and $\overline{o}$ the other order (interleaved or natural, respectively). If it corresponds to the natural order then $\Pi_O = Id$, $M_{S_O} = M_{S_1}$ and $M_{S_{\overline{O}}} = M_{S_2}$, otherwise $\Pi_O = \Pi$, $M_{S_O} = M_{S_2}$ and $M_{S_{\overline{O}}} = M_{S_1}$.

Algorithm using a hash-table:

Initialization Set $\omega_k \leftarrow 0$ for $k = 0$ to N-1

Write access update rule at half-iteration j:

Set $\omega_{\Pi_o(k)} \leftarrow \overline{\omega}_{\Pi_o(k)}$.

If $k < N\text{-}\Delta$:

Write extrinsic output $\mathbf{Z}^{(j)}_{\Pi_O(k)}$ at address $\Pi_O(k)$ of the principal memory M_P.

Else:

Write extrinsic output $\mathbf{Z}^{(j)}_{\Pi_0(k)}$ at address $\Pi_o(k)$ of M_P and at address $h\left(\Pi_o(k)\right)$ of M_{S_O}.

Read access rule at half-iteration j:

If $k \le \Delta$:

Read extrinsic input from at address $\Pi_o(k)$ of the principal memory M_P and the extrinsic input $\mathbf{Z}^{(j-3)}_{\Pi_0(k)}$ at address $h\left(\Pi_o(k)\right)$ of the secondary memory $M_{S_{\bar{O}}}$.

If bit ω_k indicates a consistency conflict, *i.e.* it has not yet been inverted during the current iteration, then provide $\mathbf{Z}^{(j-3)}_{\Pi_0(k)}$ to the processor input, otherwise provide the data of the principal memory M_P, which corresponds to $\mathbf{Z}^{(j-1)}_{\Pi_0(k)}$.

Else:

Read extrinsic input $\mathbf{Z}^{(j-1)}_{\Pi_0(k)}$ from address $\Pi_o(k)$ of the principal memory M_P.

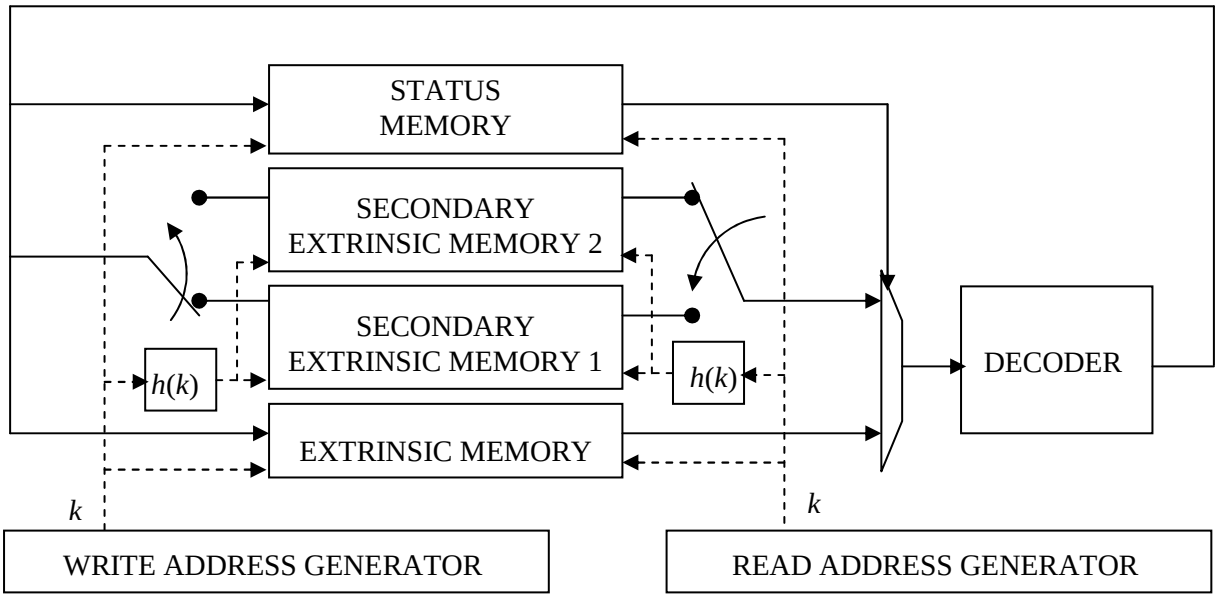


Figure 4-14: Architecture for resolving consistency conflicts comprising two secondary extrinsic memories associated with a hash function

The technique described here for a single processor can be generalized to multiple processors, with the same hash function. Both secondary memories are divided into P

memory banks with a similar memory organization as the principal memory. For an ARP interleaver, each memory bank contains the symbols, whose indices belong to the same class of congruence. Since the temporal permutation of MSTCs described in chapter 3.5 is also based on ARP interleavers, the technique can be used in a similar manner for decoding MSTCs with an overlapping of consecutive half-iterations.

Let us now study the performance of the proposed scheme.

4.4.3 Performance

In section 4.4.1 and especially in Figure 4-13 we showed that utilization of extrinsic data that are not up-to-date degrades the performance of the iterative decoder. The degradation in performance is obviously related to the proportion η of consistency conflicts among the extrinsic data. Because of the performance degradation, the proposed architectural technique must be compared to another simple solution for increasing the throughput at the cost of performance degradation. Indeed, as discussed in chapter 2.3, decreasing the maximum number of iterations enables us to increase the throughput significantly with acceptable performance degradation. The objective of the overlapping scheme is to use the activity improvement to perform more iterations, at a constant throughput. Or equivalently, for the same performance, we aim at increasing the throughput of the turbo decoder. The number of iterations allowed for the non-overlapping scheme is called the nominal number of iterations and is denoted I_n.

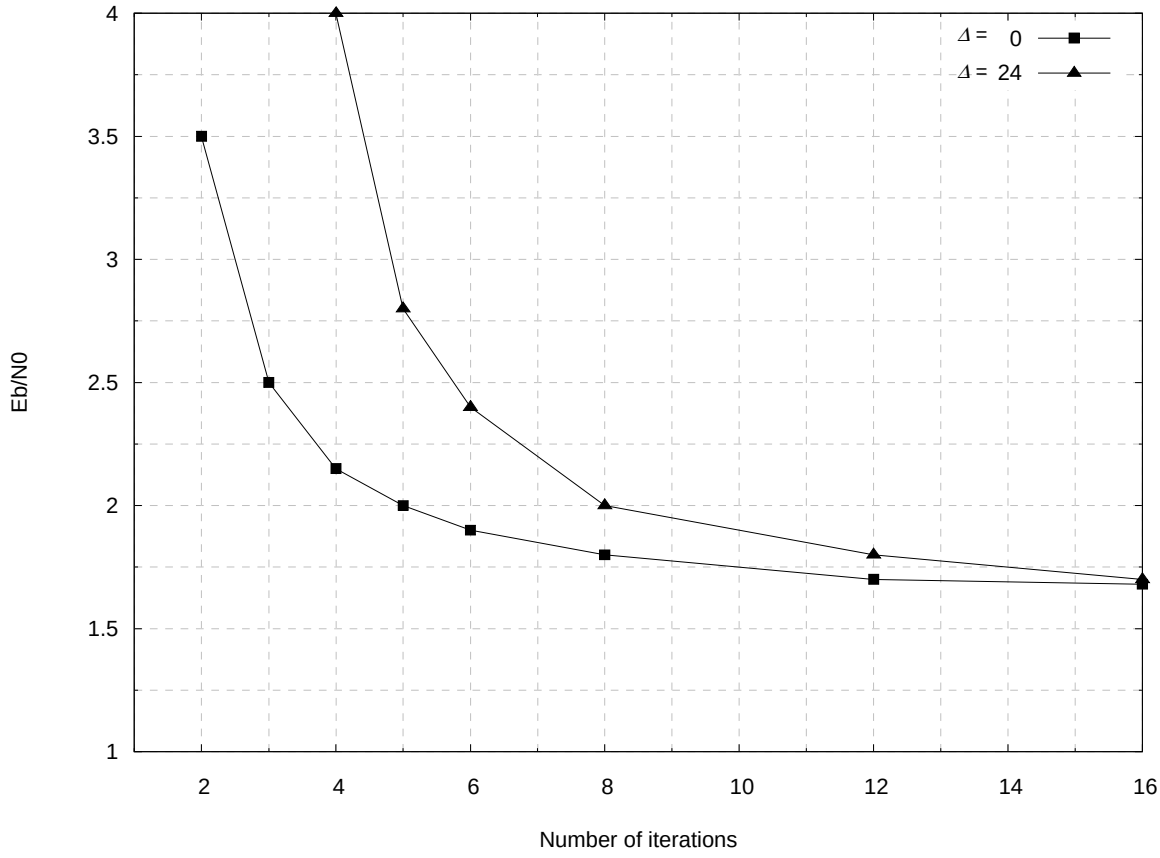


Figure 4-15: Convergence of the decoding scheme for N = 240, P = 8 and schedule $SW\text{-}\Sigma^*$

We have compared the SNR required to obtain an FER of 10^{-2} as a function of the number of iterations for classical non-overlapping decoding ($\alpha = 0.55$, $\eta = 0$) and for a scheme with a maximal overlapping depth $\Delta_{max} = L/2 + l = 24$ ($\alpha = 1$, $\eta = 0.22$). The number of consistency conflicts was obtained with the FPGA implementation presented in chapter 6 with $L = 32$ and $l = 8$, $N = 1024$ and an ARP interleaver. This turbo code is decoded with $P = 8$ processors performing schedule *OSW-*Σ^*. The results of this comparison are plotted in Figure 4-15. The comparison shows that asymptotically (at 16 iterations) both decoding schemes have equivalent performance. However, the overlapping decoding scheme converges much more slowly, in particular at the beginning of the iterative process. This comparison also demonstrates that the two schemes are equivalent in terms of performance / throughput. For instance, to achieve the target FER at $E_b/N_0 = 2$ dB, the overlapping and non-overlapping schemes require $I_{\Delta=24} = 8$ and $I_n = 4.5$ iterations respectively. The ratio of the number of iterations equals $I_n/I_{\Delta=24} = 0.56$, which equals the ratio of their respective activities. Therefore, the improvement in activity and in number of iterations of the overlapping scheme improves neither the performance, nor the throughput. Below this threshold of 2 dB, the non-overlapping scheme performs better. On the other hand, above this threshold, the two schemes are equivalent.

Since a maximal overlapping depth is not efficient, we now analyse the optimal overlapping depth that results in the best performance for a constant throughput. For this comparison, we have used the turbo decoder of chapter 6 for the WiMAX standard. Two frame sizes have been considered $N = 240$ and $N = 480$. The decoder comprises 8 processors implementing schedule *OSW-*Σ^* with $L = 32$ and $l = 8$. The clock frequency of the decoder is $f_{clk} = 80$ MHz. The comparison is conducted for two minimal throughputs $D_b = 75$ Mbits/s and $D_b = 140$ Mbits/s, the number of iterations corresponding to the maximum value that satisfies the minimal throughput. The results of this comparison are given in Table 4-1, Table 4-2, Table 4-3 and Table 4-4. From the results presented in these tables, we can make the following observations:

- the evolution in performance is not monotonic with Δ.
- the optimal overlapping depth corresponds to a low proportion η of consistency conflicts (below 1 %).
- increasing the overlapping depth and thus the activity and the number of iterations is relatively inefficient. In fact, the improvement that could be obtained with an increased number of iterations is spoiled by the increase in the proportion of consistency conflicts.
- the gain in E_b/N_0 depends on the nominal number of iterations I_n.

To illustrate these observations, we can see in Table 4-2 that the maximal gain obtained reaches almost 0.2 dB for $N = 240$ at $D_b = 140$ Mbits/s with an overlapping depth $\Delta = 15$. However, most of the gain has already been obtained for $\Delta = 8$ enabling one additional half-

iteration with no performance degradation (only a single consistency conflict). The same is observed in Table 4-1 between $\Delta = 3$ (no conflicts) and $\Delta = 7$ ($\eta = 0.02$).

Overlapping depth Δ	0	3	7	12	15	17	20	24
Activity	0.55	0.59	0.65	0.71	0.77	0.82	0.88	1
Number of iterations	4.5	5	5.5	6	6.5	7	7.5	8.5
Proportion of consistency conflicts η	-	0	0.02	0.04	0.09	0.14	0.16	0.22
E_b/N_0 for a target FER of 10^{-2}	2.4	2.31	2.30	2.31	2.35	2.36	2.33	2.29

Table 4-1: Performance at FER 10^{-2} as a function of Δ for D_b = 75 Mbit/s for a frame size N = 240 decoded with P = 8 processors performing schedule $OSW\text{-}\Sigma^*$ (L = 32, l = 8)

Overlapping depth Δ	0	8	15	20	24
Activity	0.55	0.66	0.77	0.88	1
Number of iterations	2.5	3.0	3.5	4.0	4.5
Proportion of consistency conflicts η	-	0.02	0.09	0.16	0.22
E_b/N_0 for a target FER of 10^{-2}	2.8	2.7	2.62	3.15	3.15

Table 4-2: Performance at FER 10^{-2} as a function of Δ for D_b = 140 Mbit/s for a frame size N = 240 decoded with P = 8 processors performing schedule $OSW\text{-}\Sigma^*$ (L = 32, l = 8)

Overlapping depth Δ	0	6	11	16	20	24
Activity	0.71	0.76	0.82	0.88	0.94	1
Number of iterations	6	6.5	7	7.5	8	8.5
Proportion of consistency conflicts η	-	1e-3	6e-3	0.3	0.4	0.6
E_b/N_0 for a target FER of 10^{-2}	2.1	1.95	1.93	1.92	1.92	1.92

Table 4-3: Performance at FER 10^{-2} as a function of Δ for D_b = 75 Mbit/s for a frame size N = 480 decoded with P = 8 processors performing schedule $OSW\text{-}\Sigma^*$ (L = 32, l = 8)

Overlapping depth Δ	0	6	15	24
Activity	0.71	0.77	0.88	1
Number of iterations	3.0	3.5	4.0	4.5
Proportion of consistency conflicts η	-	1e-3	0.3	0.6
E_b/N_0 for a target FER of 10^{-2}	2.38	2.18	2.20	2.18

Table 4-4: Performance at FER 10^{-2} as a function of Δ for D_b = 140 Mbit/s for a frame size N = 480 decoded with P = 8 processors performing schedule $OSW\text{-}\Sigma^*$ (L = 32, l = 8)

Let us denote Δ_{min} the maximal overlapping depth for which there exist no consistency conflicts. The performance comparison shows that most of the gain is obtained for an overlapping depth close to Δ_{min}. The technique of resolving the consistency conflict is not appropriate when the proportion of conflicts is too high (above 1 %). In this case, the decoder suffers from the huge number of consistency conflicts, which degrades the performance, especially for a high number of processors. However, we believe that this technique could be improved by using:

- an iteration-dependent overlapping depth that starts from Δ_{min} and increases along with the iterations. This makes it possible to reduce the performance degradation since it is the greatest during the first iterations.
- a different scaling factor for the extrinsic data associated with consistency conflicts. This solution makes it possible to lessen the overall impact of the lower reliability data on the iterative process.

The above-mentioned improvements are some of the short term perspectives of our work. The next section addresses another solution aiming at suppressing all consistency conflicts using a constrained interleaver design.

4.5 Joint interleaver-architecture design for activity increase

In the previous section, we handled the consistency conflicts during the execution of the algorithm. Unfortunately, the results are, so far, not very satisfactory. For this reason, when the designer has the freedom to modify the interleaver, we propose to design the interleaver so that it prevents the appearance of any consistency conflicts when overlapping of consecutive half-iterations is used. Our methodology is based on a multi-level hierarchical interleaver associated with MSTCs, but also with conventional SSTCs.

This methodology is recent and, until now, we have not exploited all its possibilities nor studied it thoroughly. For this reason, throughout this chapter, we present some simple, yet instructive, examples for designing constrained interleavers. For the sake of simplicity, we will use schedule *OSW-*Σ^{-} for our constrained design. However, in order to draw fair comparisons, they will be compared to non-constrained interleavers optimized for performance and decoded with schedule *SW-*Σ^{*}, which is always more efficient than *SW-*Σ^{-} (see section 4.1.3).

After a brief overview of the joint interleaver-architecture concept, we introduce a graphical representation that illustrates the constraints imposed on the interleaver design. Then, we use this representation to design a simple turbo code decoded with overlapping depth $\Delta = L$ that demonstrates the potential of our methodology. After describing our multi-level hierarchical interleaver, we use its properties to design another turbo code, decoded with a maximal overlapping depth $\Delta = L+l$. Finally, we make the generalization to other schedules and other numbers of processors.

4.5.1 Overview

The solution for suppressing all consistency conflicts relies on constraints imposed on the interleaver design. For a given schedule, and a fixed overlapping depth, the interleaver maps symbols used during the overlapping region at the beginning of a half-iteration to symbols, whose extrinsic information has already been produced during the preceding half-iteration. The interleaver must satisfy this property for both overlapping regions $\Omega_{i\rightarrow n}$ and $\Omega_{n\rightarrow i}$.

The constraints imposed on the interleaver design introduce the existence of banned regions in the interleaver design. Indeed, a symbol that is read at the beginning of the half-iteration, is not allowed to be interleaved to a symbol written during the last cycles of the preceding iteration. The next section introduces a graphical representation of these banned regions.

4.5.2 Graphical representation of the banned regions

Let us use a two-dimensional representation of an interleaver Π introduced in [Heegard *et al*.-99]. The natural addresses are represented on the *x*-axis, whereas the interleaved addresses are represented on the *y*-axis. Then the symbol with index *y* that is interleaved to the index *x*= Π(*y*) of the natural order is represented by a point with coordinates (*x*,*y*).

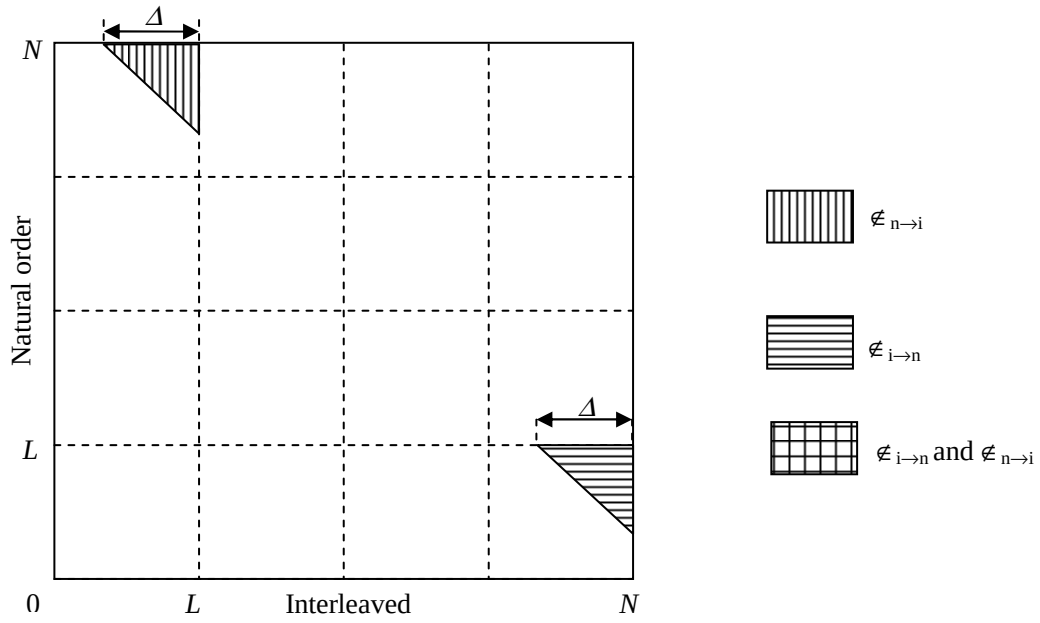


Figure 4-16: Banned regions for $P = 1$ processor using schedule $OSW\text{-}\Sigma^{-}(\Delta)$, $\Delta \le L$

The banned regions associated with a given schedule Σ can be represented on this graphical representation of the interleaver. For instance, Figure 4-16 represents the banned regions for a single processor architecture with schedule $OSW\text{-}\Sigma^{-}(\Delta)$, $\Delta \le L$. The banned region corresponding to the transition $T_{n\to i}$ is denoted $\notin_{n\to i}$, while the other corresponding to the transition $T_{i\to n}$, is denoted $\notin_{i\to n}$. It is assumed that the reading of the extrinsic inputs starts at time $\tau = 0$ and is performed window by window according to the backward order, *i.e.* decreasing order. The first extrinsic outputs are written into the memory at time $\tau = L + l$. They are written window by window in the increasing order. The first data read at time $\tau = 0$ cannot belong to one of the symbols generated by the preceding half-iteration during the last Δ cycles. At time $\tau = \tau'$ the excluded region is reduced to $\Delta\text{-}\tau'$ positions, and for $\tau = \Delta$, all symbols are permitted. This explains the upper triangle shape of $\notin_{i\to n}$ in the first window associated with the transition $T_{i\to n}$. For transition $T_{n\to i}$, the banned region $\notin_{n\to i}$ also corresponds to an upper-triangle shape.

This graphical representation is called the mask of the interleaver denoted $\in$, since the points defining the interleaver function can be placed only on regions that are not banned. It shows that using a single processor architecture, for moderate overlapping depth, it is not a complicated task to design an interleaver suited to this mask. In fact, a large number among the $N!$ different interleavers are allowed. From this graphical representation, one can also find the average number of consistency conflicts for a uniform random interleaver. In fact, under

this assumption, the dots representing the interleaver are distributed uniformly over the square of size $N \times N$. The number of consistency conflicts corresponds to the number of dots in the banned regions, *i.e.* Δ^2 / N. For symbols belonging to the two banned regions, two consistency conflicts are counted: one for each transition.

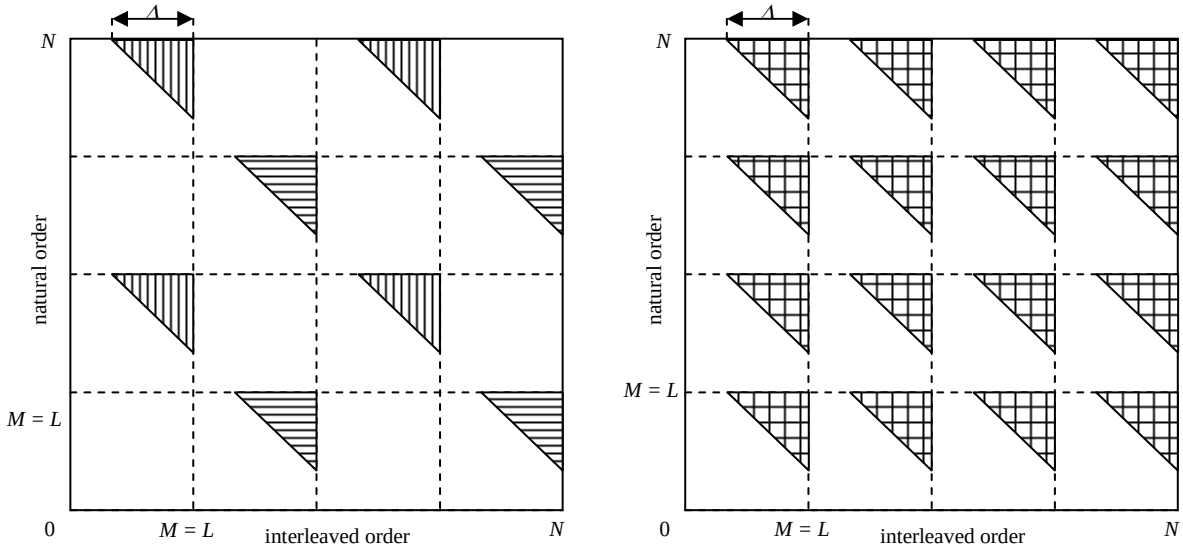


Figure 4-17: Interleaver mask for P processors using schedule $OSW\text{-}\Sigma^{-}(\Delta)$, $\Delta \leq L$: a) $P = 2$; b) $P = 4$ (the same caption as in Figure 4-16 is used)

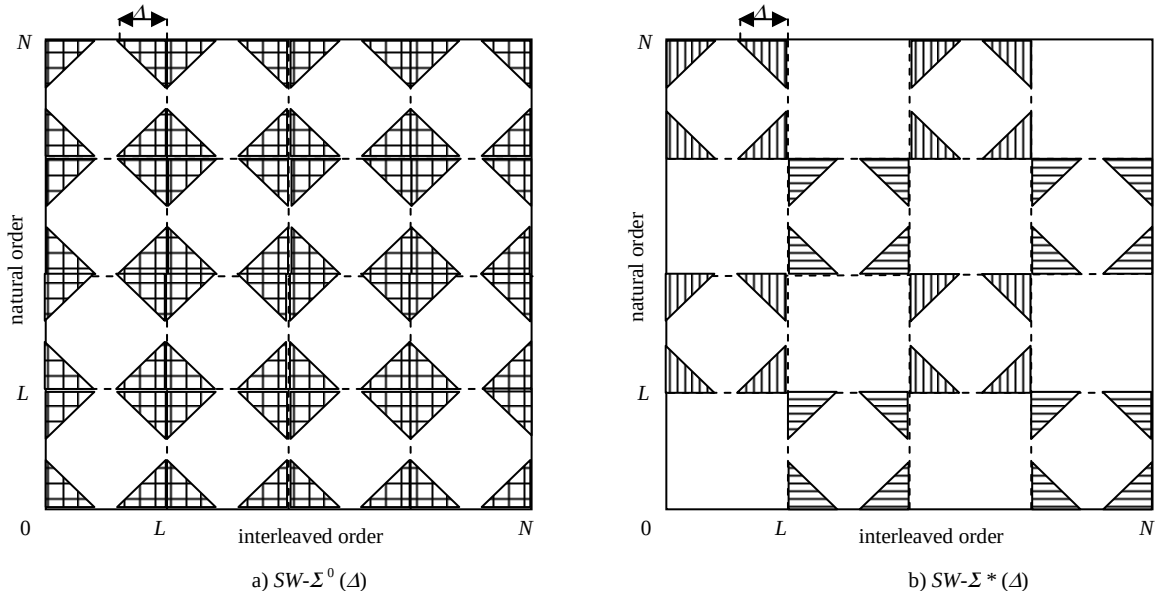


Figure 4-18: Interleaver mask for $P = 4$ processors with $\Delta \leq L/2$: a) schedule $OSW\text{-}\Sigma^{0}(\Delta)$; b) schedule $OSW\text{-}\Sigma^{*}(\Delta)$ (the same caption as in Figure 4-16 is used)

Figure 4-17 represents the mask for a P-processor architecture using schedule $SW\text{-}\Sigma^{-}$ ($\Delta \leq L$). It shows that increasing the number of processors increases the area of the banned regions: if P is multiplied by 2, the number of banned triangles is multiplied by 4. This result is coherent with the average number of consistency conflicts with P processors, which equals $\Delta^2 \cdot P^2 / N$.

But the major problem concerns the drastic reduction in available area for placing the N dots defining the interleaver. In fact, the maximum number of distinct interleavers that are allowed by this mask is greatly reduced, which leads to a reduction in the maximal spread of an interleaver.

In fact, using a volume argument, it has been shown in [Boutillon *et al.*-05] that the maximal spread of an interleaver is bounded by the sphere bound S_B. The sphere bound was obtained by considering the following condition. If the spread of the interleaver is S, then on the 2-dimensional representation of the interleaver, each dot has no neighbours with a sphere of radius $S/2$ for the l_1-norm. In the two-dimensional case, this sphere corresponds to a diamond. The sphere bound is obtained when the volume of the N spheres totally fills the volume of the space, as shown in Figure 4-19. For more details and a geometrical demonstration of the computation of the sphere bound, we refer the reader to Appendix B.

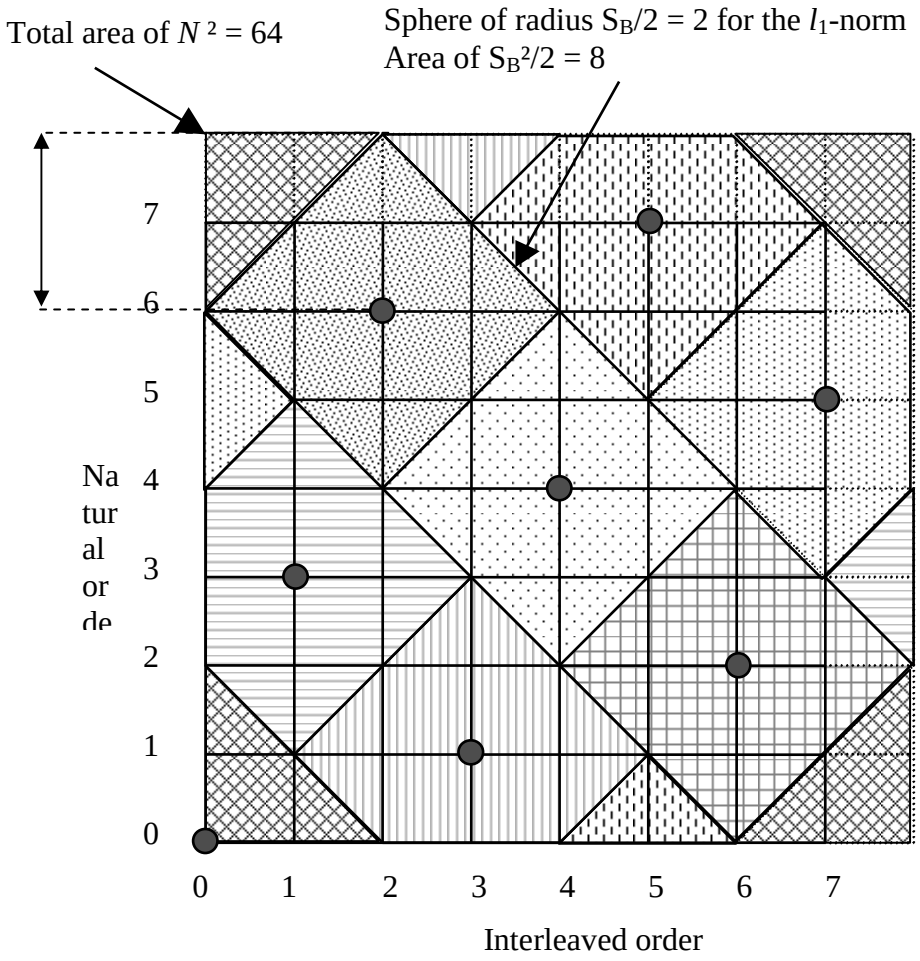


Figure 4-19: Two-dimensional representation of an interleaver with spread $S_B = \sqrt{2N}$ for $N = 8$

Using a similar volume argument, it is clear that increasing the area of the banned region decreases the maximal spread of an interleaver satisfying the mask. For this reason, it is not an easy task to design an interleaver satisfying the mask corresponding to the decoding architecture and schedule with good performance. Due to the reduced spread, the resulting interleaver will undoubtedly have a poor convergence property.

In order to relax the spread constraint on the interleaver, Multiple Slice Turbo Codes are used. In fact, as has already been discussed in chapter 3.4, the spread of an MSTC constructed with P slices of M symbols with $P \geq M$ is infinite. The next section presents a simple example of a mask-constrained interleaver design using MSTCs.

4.5.3 A simple example

In this example, we design a Multiple Slice turbo code with $K = P = M$ slices of M symbols. The MSTC is decoded using schedule $OSW\text{-}\Sigma^{-}(\Delta)$ with window size $L = M$. For this design, we use a hierarchical interleaver of parameter P (see chapter 3.3), denoted HI(P), that satisfies the corresponding mask. The same interleaver HI(P) will be used to design a SSTC that thus can be decoded in parallel with the same schedule $OSW\text{-}\Sigma^{-}(\Delta)$. Then, we will compare the efficiency of our approach by comparing the two codes to a conventional turbo code with ARP interleaver.

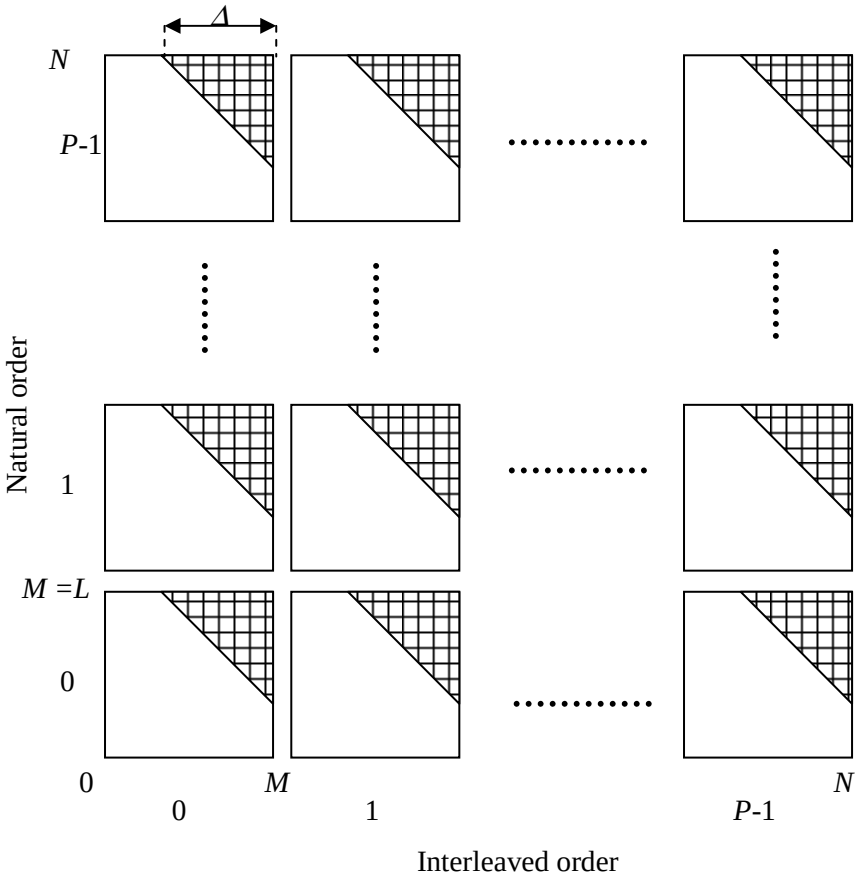


Figure 4-20: Interleaver mask for an MSTC decoded with P processors performing schedule $SW\text{-}\Sigma^{-}$ ($L = M$) with overlapping depth $\Delta < L$

The mask for the interleaver design corresponding to a P-processor architecture is depicted in Figure 4-20 for an overlapping depth of $\Delta < L$. Considering a single identical temporal permutation Π_T for all slices, the interleaver mask can be transposed on the temporal permutation. This simplified representation on the temporal permutation only will be used in Figure 4-21.

For $\Delta = L$, the banned regions do not totally fill the interleaver space. But no temporal permutation satisfies this mask, since there exists at least one symbol (symbols with indices M-1 mod M in the interleaved order) for which no interleaving point is available. The mask is said to be closed. The extreme case $\Delta = L$-1 leaves open the interleaver mask as shown in Figure 4-19. Let us construct the temporal permutation recursively. For index $t = M$-1, the only available position is $\Pi_T(t) = 0$. Then, for $t = M$-2, two interleaved indices are allowed $\Pi_T(t) = 0$ and $\Pi_T(t) = 1$. Since $\Pi_T(t) = 0$ has already been used for $t = M$-1, only $\Pi_T(t) = 1$ suits. A recursive computation of the interleaved indices leads to the reverse order permutation represented in Figure 4-21. When decoding the frame in the interleaved order, the backward recursions require symbols with interleaved indices 0 to M-1, produced in this order by the forward recursions working in the natural order. The same arrangement holds for transition $T_{i\to n}$. For this mask, the extrinsic output produced by the current half-iteration are immediately used as the extrinsic input of the next half-iteration.

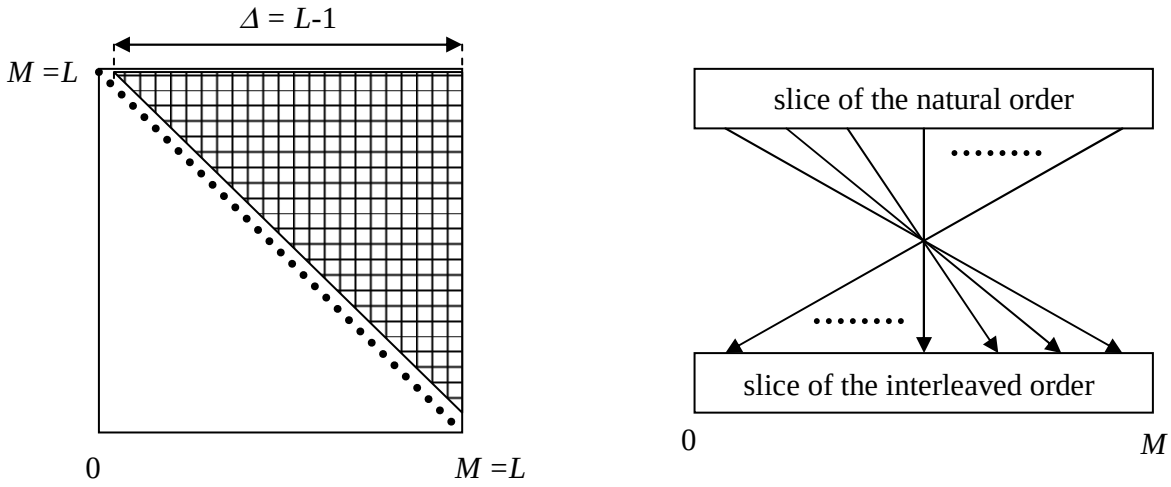


Figure 4-21: Temporal permutation mask for $\Delta = L$-1 and reverse order permutation

With this temporal permutation, the only degree of freedom to design the interleaver and optimize its performance is spatial permutation. In fact, because of the slice property, the spread of the interleaver is infinite and there exists no primary cycle in the interleaver. The objective of the spatial permutation is thus to increase the length of the secondary cycles. Similarly to the spatial permutation design of chapter 3.5, an irregular spatial permutation is chosen here.

For example, for a frame size N = 1024 double-binary symbols, an MSTC is built with 32 slices of 32 symbols. It uses a reverse order temporal permutation associated with the irregular spatial permutation

$$\Pi_S = \{0, 2, 24, 1, 17, 27, 6, 23, 15, 4, 18, 26, 12, 21, 8, 29, 11, 5, 9, 13, 28, 16, 3, 31, 22, 14, 19, 10, 7, 25, 20, 30\}. \quad \textbf{(4-19)}$$

This turbo code designed here can be decoded in parallel with up to P = 32 processors performing schedule $OSW\text{-}\Sigma^{-}(L\text{-}1)$. The same interleaver is also used to design a SSTC that is also decoded with P = 32 processors performing the same overlapping schedule. The two turbo codes are compared to a conventional turbo code designed with an ARP interleaver with

parameters $\lambda = 45$ and $\mu = \{4,20,52\}$. In order to draw fair comparisons, this latter is decoded using a parallel schedule $SW\text{-}\Sigma^{*}$ with $P = 32$ processors (this schedule has a higher activity than $SW\text{-}\Sigma^{-}$).

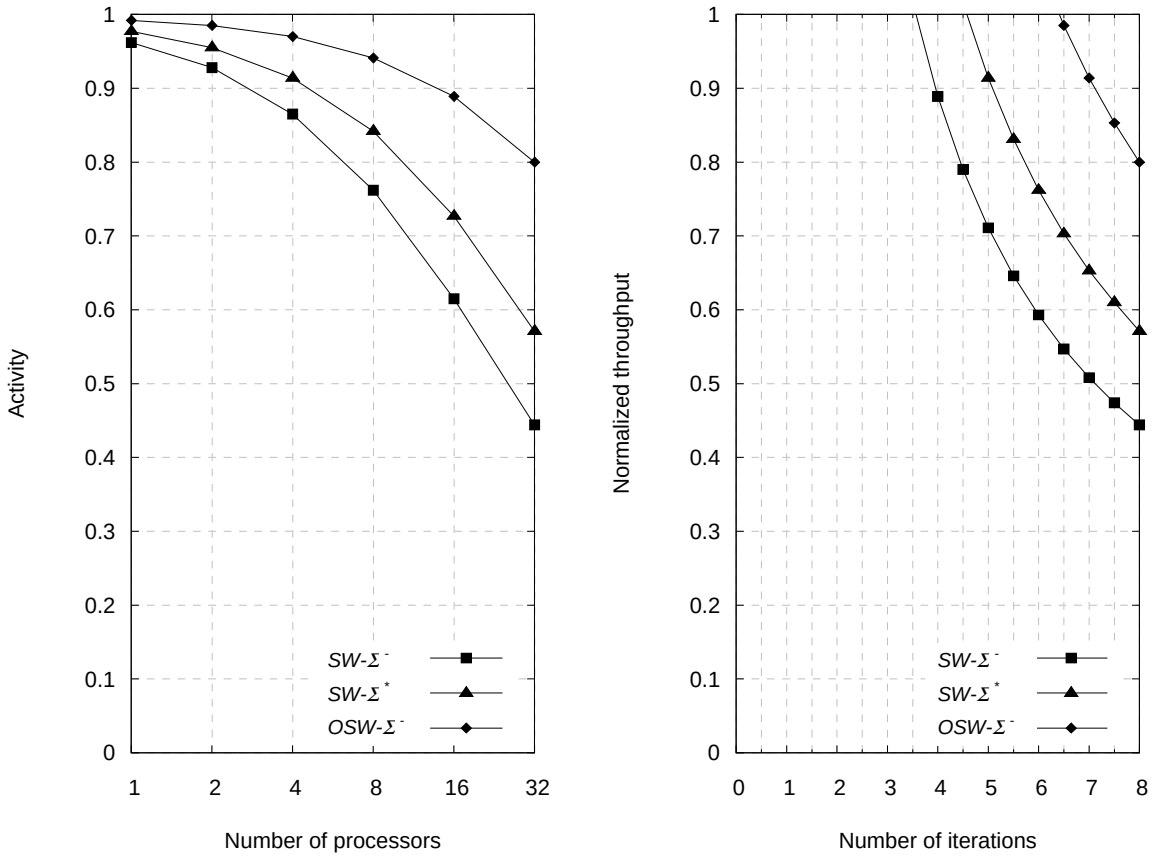


Figure 4-22: Activity as a function of the number of processors for $N = 1024$, $L = 32$, $I = 8$ and normalized throughput for $P = 32$ as a function of I_{max}

Let us now analyze the improvement in activity obtained with the overlapping schedules and compare it to schedule $SW\text{-}\Sigma^{*}$. The activities as a function of the number of processors are shown in Figure 4-22. For $P = 32$ processors, the activity of schedule $SW\text{-}\Sigma^{-}$ is improved thanks to an overlapping depth of $\Delta = L\text{-}1$ from 0.44 to 0.8, which represents almost a multiplication by 2. Moreover, compared to schedule $SW\text{-}\Sigma^{*}$, whose activity is equal to 0.57, this represents an improvement of 40 % of the activity. The performance comparison of the three codes and associated schedules needs to be performed at equivalent throughput for $P = 32$ processors. Figure 4-22 compares the normalized throughput of the schedules for $P = 32$ processors as a function of the number of iterations I_{max}. The throughput is normalized relatively to the throughput that an architecture with $P = 32$ processors and an activity of 1 would achieve after $I_n = 8$ iterations. To achieve a normalized throughput of 0.8, the two constrained codes can be decoded with 8 iterations of $OSW\text{-}\Sigma^{-}(L\text{-}1)$, compared to only 4.5 with $SW\text{-}\Sigma^{-}$. The decoder for the code with an ARP interleaver can perform only 5.5 iterations of $SW\text{-}\Sigma^{*}$.

The simulated performance of the three codes is shown in Figure 4-23 (right) for a Max-Log-MAP algorithm with input quantization on w_d = 6 bits. Surprisingly, the two constrained codes perform equivalently and have very good performance, although the error floor appears earlier than for the turbo code with ARP interleaver. At an equivalent throughput, the additional number of iterations allowed by the constrained interleaver design yields a performance improvement of 0.25 dB at an FER of 10^{-3}. In Figure 4-23 (left), we have plotted the SNR required for the two interleavers ARP and HI(32) (associated with a MSTC) to achieve an FER of 10^{-3}. To achieve a target E_b/N_0 of 1.7 dB for the three codes, the number of iterations of the decoder for the MSTC can be reduced to 6.5. This reduction in the number of iterations leads to a normalized throughput of 1, which represents an improvement of 25 %. These first promising results demonstrate the interest of our methodology.

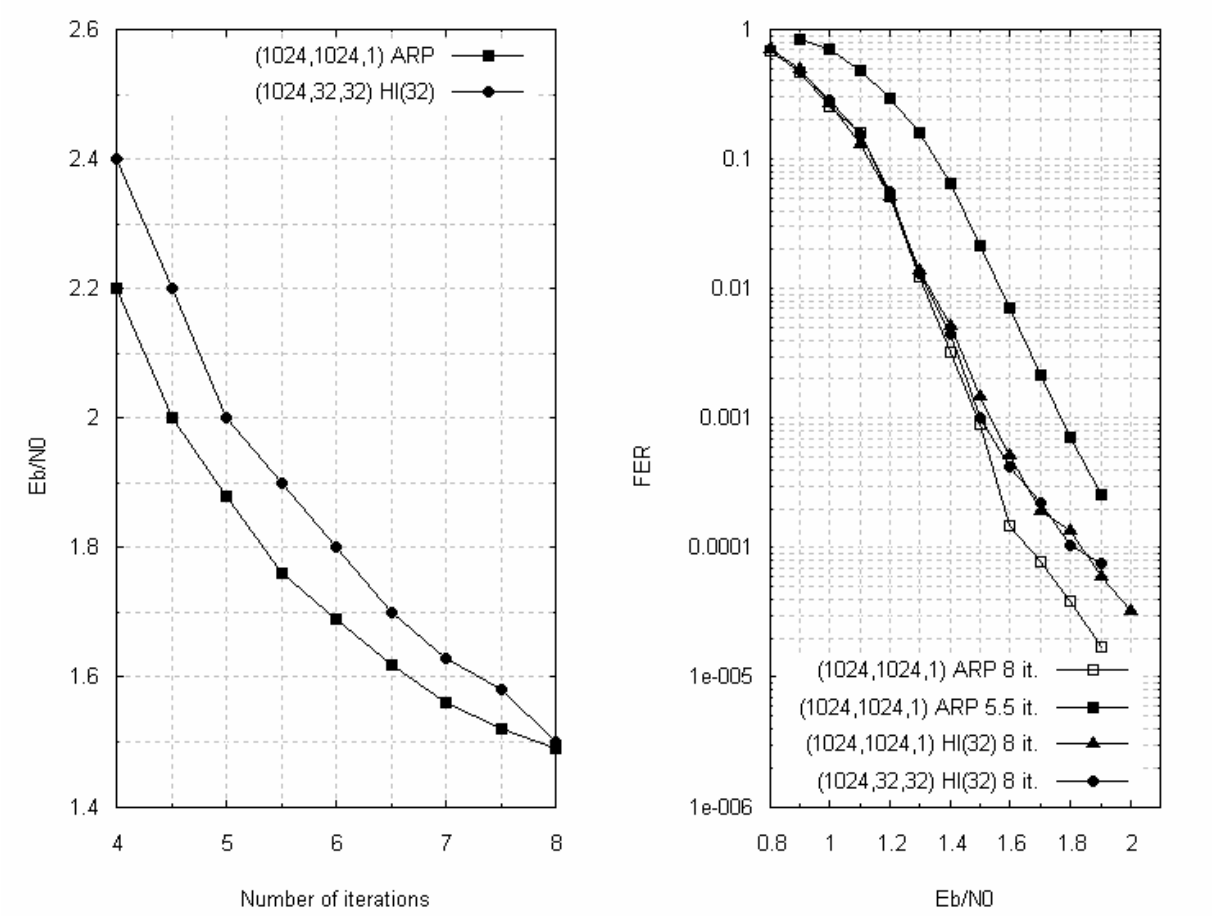


Figure 4-23: Performance comparison of turbo codes of size N = 1024 decoded with P = 32 processors

If the MSTC is decoded with fewer than 32 processors, the interleaver is obviously over-constrained, particularly the temporal permutation. Based on this example, the constraints on the temporal permutation will be relaxed using a hierarchical interleaver approach presented in the next section.

4.5.4 Hierarchical interleaver approach

Let us consider an architecture comprising P processors decoding in parallel an MSTC, composed of K slices of M symbols. Assuming a sliding window algorithm with window size

L, the banned regions of the interleaver correspond to the intersection between the last and first windows of the two dimensions. Therefore, with an overlapping depth $\Delta \leq L$, the interleaver mask imposes constraints on the first *L* and last *L* symbols of the temporal permutation. In order to take into account this locality of the banned regions, the following two are used (see Figure 4-24):

- each slice is again divided into *Q* distinct slices, where *Q* corresponds to the number of windows for the sliding window schedule over the previous slice of size *M*, $Q = \lceil M / L \rceil$.
- the temporal permutation is split into two levels: the first level Π_T^1 shuffles the data between slices, while the second Π_T^2 shuffles the data within slices.

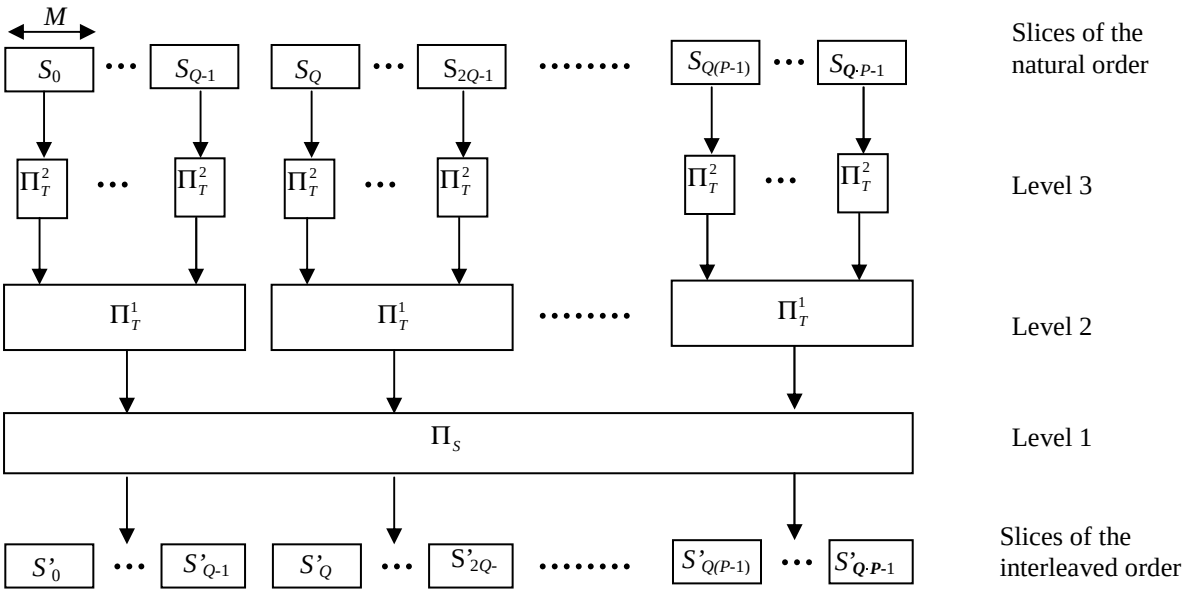


Figure 4-24: Hierarchical interleaver with three levels of permutations for an MSTC with $K = P \cdot Q$ slices of M symbols

This additional level in the hierarchical interleaver is used to facilitate the design of an interleaver satisfying the interleaver mask. Moreover, similarly to the parallel interleaver construction, slicing into *Q* smaller slices makes it possible to easily design the two levels of the temporal permutation. In fact, thanks to slicing, two symbols that are not in the same slice do not share any direct relation. In conclusion, the hierarchical interleaver denoted HI(*P*, *Q*) is composed of three levels:

- the first level Π_S, called the spatial permutation, which handles the decoding parallelism. It shuffles the data between slices of distinct memory banks.
- the second level Π_T^1, belonging to the temporal permutation, shuffles the data between slices of the same memory banks.
- the third level Π_T^2, belonging to the temporal permutation, shuffles the data within slices.

The last two levels are constrained to satisfy the interleaver mask, in order to enable a parallel decoding with P processors and an activity close to 1. In the first step, we consider that the total number of slices $K = P \cdot Q$ is greater than the number of symbols per slice M. Spreading the M symbol of a slice over the K slices of the other dimension yields an interleaver that relaxes the spread constraint (the spread equals to $2M$, as shown in chapter 3.4 and Appendix B). In most cases, the constraints resulting from the interleaver mask are supported both by permutations Π_T^1 and Π_T^2. If K is much greater than M, then the symbols are spread over the slices that do not belong to the banned regions. In this case the constraints are transferred to permutation Π_T^1 only.

Each of the P processors performs a sliding window algorithm over the Q slices it decodes in each dimension. Let S_{pQ+r} and S'_{pQ+r}, $0 \le r \le Q\text{-}1$ denote the slice r of the natural and interleaved order, respectively, decoded by processor p. Each slice of size M is decoded with a classical non-windowed schedule. However, the same processor decodes Q slices successively in the same dimension with an overlapping in the processing of adjacent slices. Therefore, the schedule for decoding the Q slices corresponds to a windowed schedule with window size M. The only difference is that during the same half-iterations, the state metrics are not transmitted from one window to the next one: the forward and backward recursions are initialized independently for each slice.

Since the banned regions are identical for groups of $Q \times Q$ slices, the design of the constrained interleaver is performed on the mask restricted to the two levels of the temporal permutation. Moreover, the notation of the Q slices considered in each dimension is simplified to S_r and S'_r, $0 \le r \le Q\text{-}1$.

The design of a constrained hierarchical interleaver using MSTCs is illustrated with a few examples presented hereafter.

Example 4-2:

The mask of the temporal permutation for an MSTC with parameters $N = 1024$, $P = 16$, $Q = 2$, $K = M = L = 32$ is represented in Figure 4-25. For this mask it is considered that in each dimension, the processor p decodes the slices S_0 and S_1 (or S'_0 and S'_1) successively.

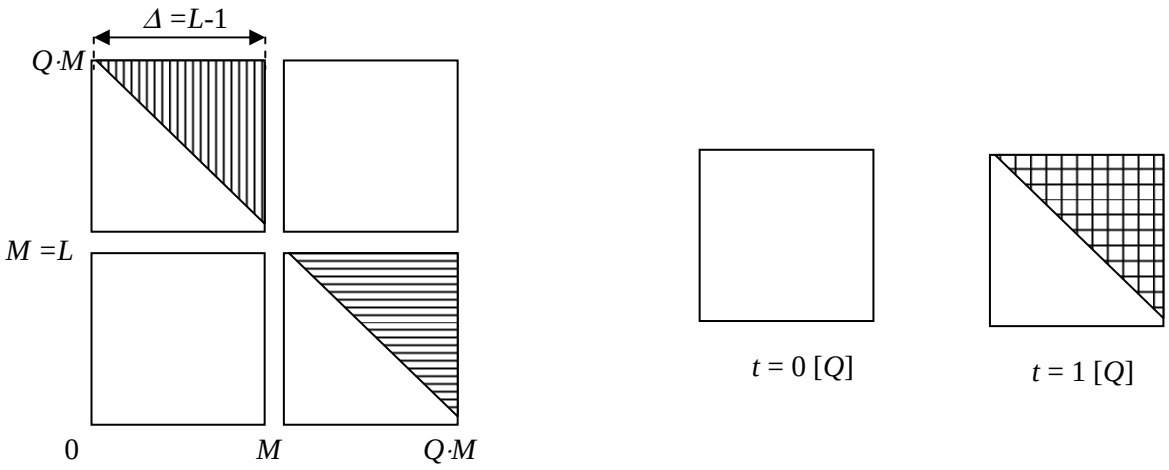
The first level of the temporal permutation Π_T^1 spreads the symbol of one slice on the $Q = 2$ other slices of the other dimension according to the equation of a circular rotation of amplitude A':

$$\Pi_T^1(t, r) = A'(t \bmod Q) + r \bmod Q \qquad \textbf{(4-20)}$$

where t is the index of the symbol within the slice and the $A' = \{0,1\}$.

For the second level, an identical permutation Π_T^2 is used for all slices. This enables us to drastically reduce the memory required to store the interleaver, and simplifies the design task. The constraints resulting from the temporal permutation mask are then split into Q sub-masks applied to the classes of congruence modulo $Q = 2$, *i.e.* the classes of even and odd symbols,

as represented in Figure 4-25.b. For transition $T_{n\to i}$, the resulting banned region $\notin_{n\to i}$ corresponds to the upper triangle of the intersection between slices S'_0 and S'_1. The constraints are applied to the odd indices of the temporal permutation. For the reverse transition $T_{i\to n}$, the banned region $\notin_{i\to n}$ corresponds to the upper triangle of the intersection between slices S'_1 and S_0. The same constraints are again applied to the odd indices. The temporal permutation mask imposes all the interleaver constraints on the odd indices, and no constraints on the even indices. The same interleaver as in 4.5.3 can be used, with almost the same performance.



a) Mask of the temporal permutation

b) Separation for the classes of congruence modulo Q

Figure 4-25: Mask of the temporal permutation for MSTC (*N* = 1024, *M* = 32, *P* = 16, *Q* = 2) decoded by *P* = 16 processors using the schedule *OSW-Σ⁻(L-1)*

So far, we have considered the case of overlapping depth $\Delta \leq L$. But the activity can be increased further if the overlapping depth is raised to its maximal value $\Delta_{max} = L + l$. The next section presents this case.

4.5.5 Interleaver design with a maximal overlapping depth

In this section we consider turbo codes of size N decoded with P processors, each decoding a block of $2L$ symbols. When $\Delta_{max} = L + l$, $\Omega_{i\to n}$ and $\Omega_{n\to i}$ overlap on two windows (or slices): the last and the first two windows (slices) in each dimension. As a consequence, the areas of the corresponding banned regions $\notin_{i\to n}$ and $\notin_{n\to i}$ are increased. The impact on the banned regions and on the interleaver constraints is analyzed in the following example.

Example 4-3:

Let us consider the MSTC code constructed in Example 4-2, decoded using schedule *OSW-Σ⁻*$(L+l)$. The corresponding banned regions are depicted in Figure 4-26. This mask shows that at the beginning of the half-iteration, the symbol read can correspond to symbols from a single slice of the other dimension. Therefore, in order to satisfy the mask, a three-level hierarchical interleaver HI(16,2) is designed (Q = 2). The two levels of the temporal permutation are chosen as follows. The first level Π_T^1 is given by:

$$\Pi_T^1(t,r) = A'(t \bmod M) + r \bmod Q \quad \textbf{(4-21)}$$

where A' is chosen to be as irregular as possible:

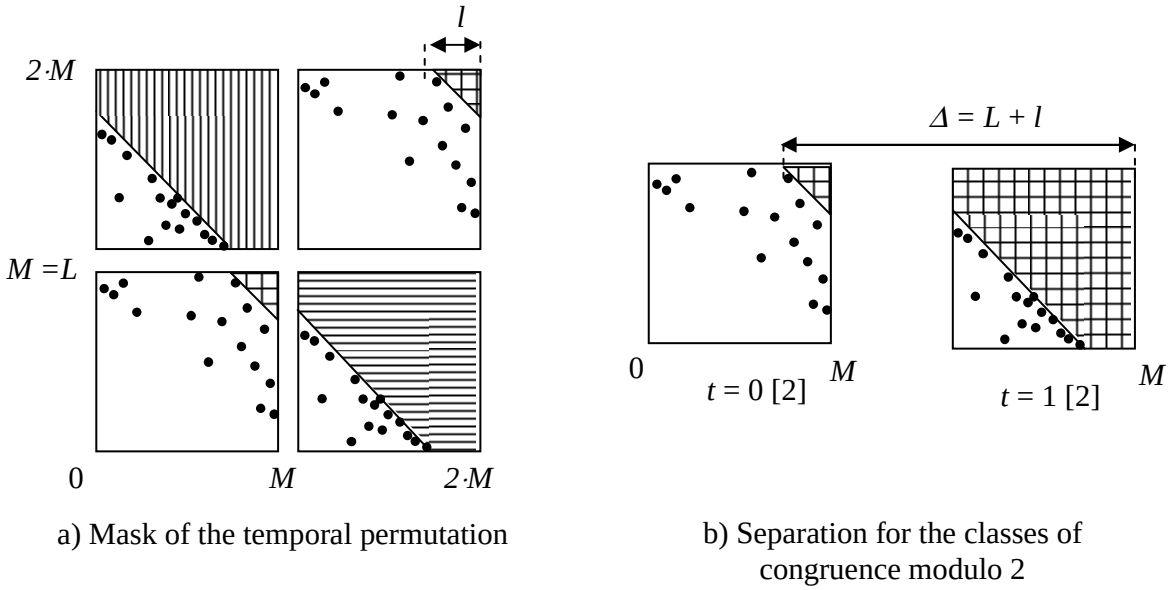
$$A' = \{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1\} \quad \textbf{(4-22)}$$

The second level of the temporal permutation is chosen as:

$$\Pi_T^2(t) = \{22, 28, 21, 27, 12, 29, 18, 25, 1, 14, 11, 5, 9, 10, 4, 7, 24, 6, 31, 3, 17, 2, 23, 0, 30, 19, 26, 16, 8, 20, 13, 15\}, \quad \textbf{(4-23)}$$

and, the irregular spatial permutation used is:

$$\Pi_S = \{0, 2, 8, 6, 15, 4, 12, 2, 11, 5, 9, 13,3, 14, 10, 7\}. \quad \textbf{(4-24)}$$



a) Mask of the temporal permutation

b) Separation for the classes of congruence modulo 2

Figure 4-26: Temporal permutation mask with $\Delta = \Delta_{max} = L + l$ and corresponding interleaver

Using the hierachical interleaver HI(16,2) described above, we design two codes, the first one being an MSTC with 32 slices, and the second being a SSTC. They are decoded with $P = 16$ processors with schedule $OSW\text{-}\Sigma^{-}(24)$. They are compared to the same SSTC with an ARP interleaver presented in section 4.5.3. The activities and normalized throughputs for the three corresponding decoders are compared in Figure 4-27. To achieve a normalized throughput of 1, the decoder for the constrained codes can perform 8 iterations of schedule $OSW\text{-}\Sigma^{-}$ (activity 1), whereas the decoder for the code with ARP interleaver is limited to 5.5 iterations of $SW\text{-}\Sigma^{*}$ (activity 0.72).

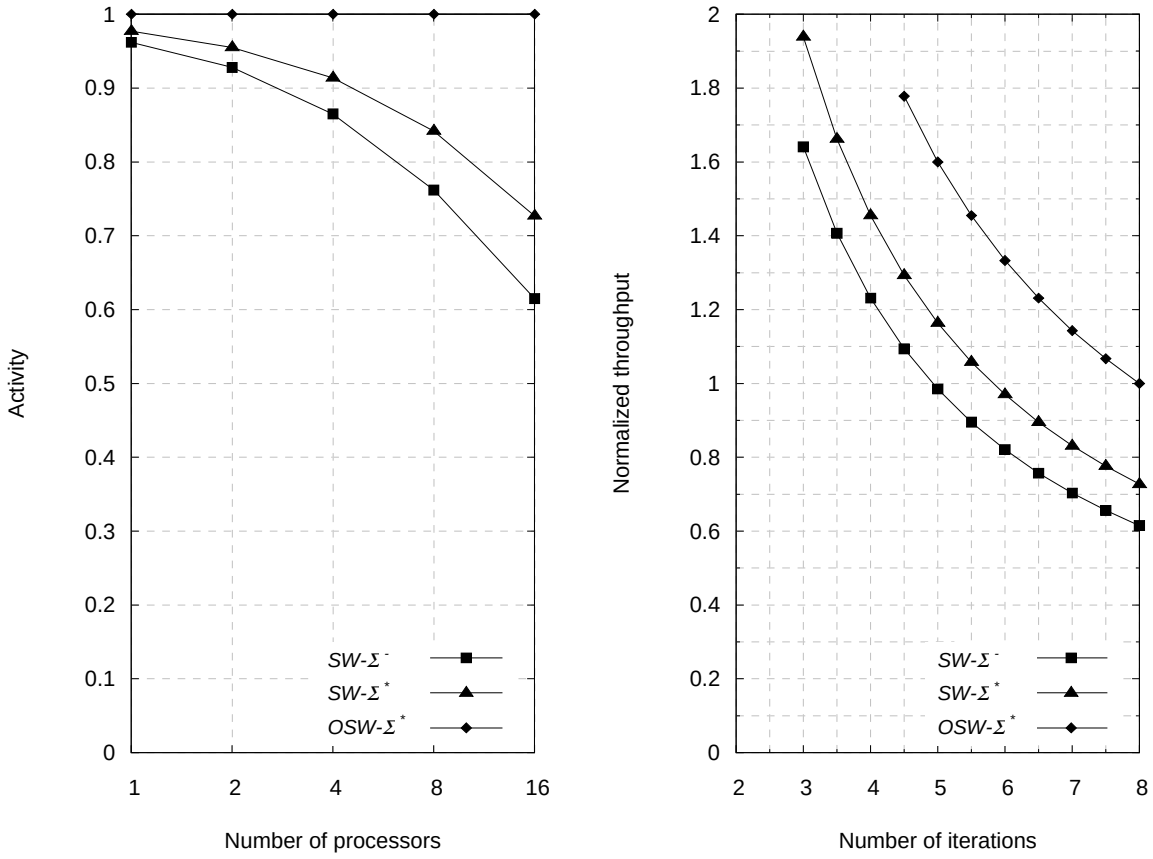


Figure 4-27: Activity as a function of the number of processors for N = 1024, L = 32, l = 8 and normalized throughput for P = 16 processors as a function of I_{max}

The error decoding performance of the codes is compared in Figure 4-28. It shows that the turbo code constructed with slices has a lower error floor compared to the SSTC with the same interleaver. Compared to conventional turbo coding and decoding, the constrained interleaver associated with MSTC yields an improvement in E_b/N_0 of 0.15 dB down to an FER of 10^{-4}.

In Figure 4-28 (left), we have plotted the SNR required for both codes to achieve an FER of 10^{-3}. If the number of iterations of the decoder for the MSTC HI(16,2) is reduced to 6.5 then the two codes have equivalent performance. This reduction in the number of iterations leads to a normalized throughput of 1.25, which represents an improvement of 25 %. The MSTC turbo code described in section 4.5.3 is also represented in this figure. The curves show clearly that relaxing the constraints imposed on the interleaver HI(32) designed in 4.5.3 enables an improvement in FER.

This example shows the interests of MSTCs associated with a hierarchical interleaver for designing interleavers with a low error floor. With the slice approach, it is easier to break the low-weight patterns than with a single slice and the same interleaver. Moreover, we believe that the error floor of the MSTC observed in this example could be lowered by studying the low weight error patterns in a similar manner to what was presented in chapter 3.4.

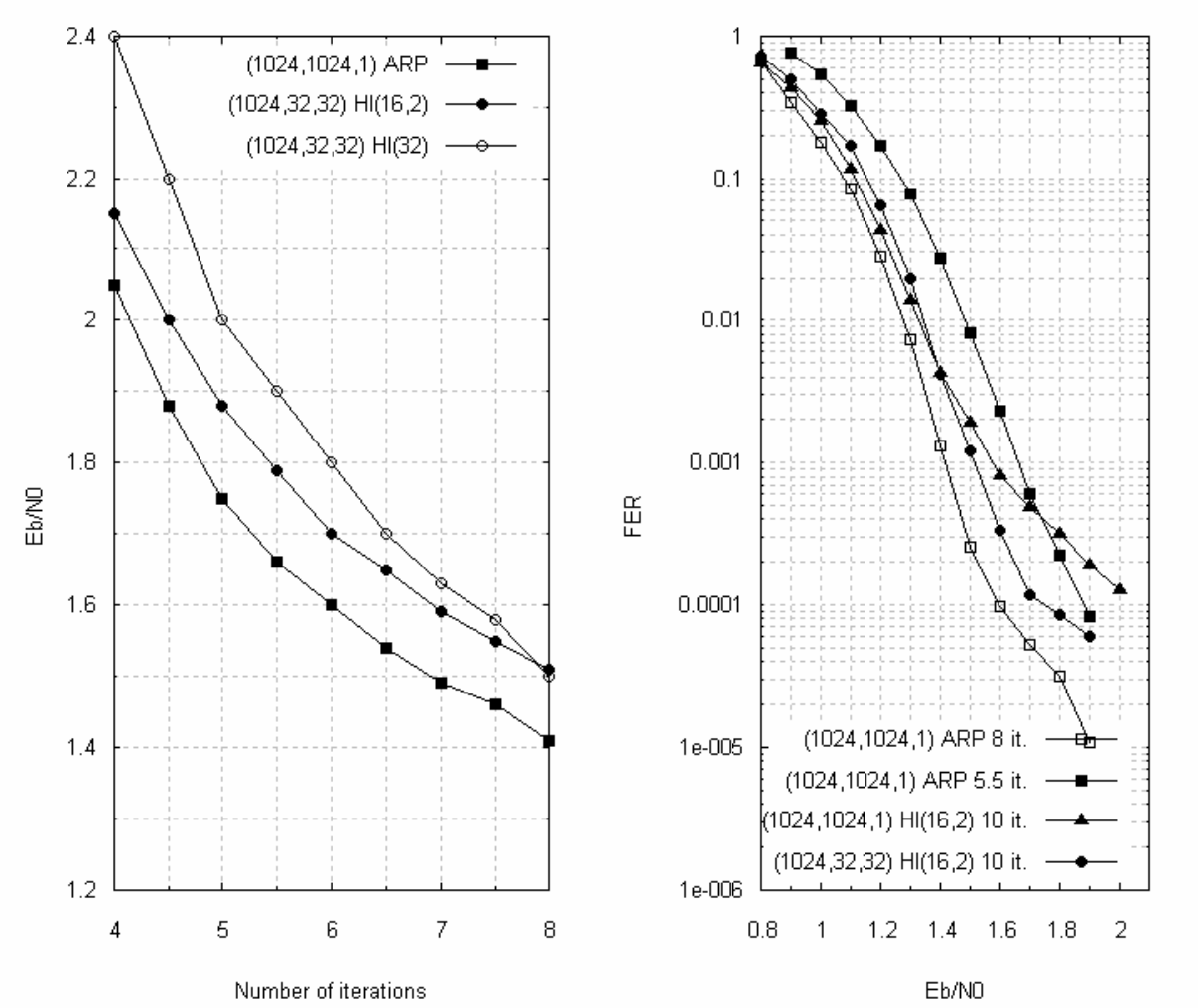


Figure 4-28: Performance comparison of the turbo codes of size N = 1024 decoded with P = 16 processors

4.5.6 Generalization and discussion

As a general rule, the design technique is efficient for long frame sizes and/or a small number of processors, since it faces the problem of the size of the slices, which becomes too small. In fact, the constraints $P \geq M$ can be relaxed, but in this case, the spread of the interleaver is no longer infinite. As a consequence, the temporal permutation and particularly its second level Π_T^2 must be designed regarding the constraint of spread.

The technique of a joint interleaver-architecture design has been experimented with MSTCs and SSTCs associated with multi-level hierarchical interleavers. First results are very promising with MSTCs and have shown that efficient interleavers can be designed easily allowing a maximal overlapping depth, which results in an activity close to 1. Yet, important work on these interleaver constructions still remains, in particular in the study of the distribution of low-weight codewords. In fact, the drawback of this interleaver construction, where the temporal permutation is almost chosen at random, is that it does not enable a simple study of the low-weight codewords. In association with a more structured interleaver, a comprehensive study of the weight codewords, and particularly the Secondary Error Patterns

associated with secondary cycles (see chapter 3.4) might improve the asymptotic gain of these codes.

4.6 Conclusion

In this chapter, we have addressed several solutions for retrieving the loss of activity induced by parallel decoding architectures. We have considered several aspects in the design space, and for each aspect, we have proposed a technique resulting in activity improvement.

At the lowest level, the only degree of freedom lies in the choice of the forward-backward schedule. Therefore, when the code is fixed, we have proposed an optimized schedule for parallel decoding denoted $SW\text{-}\Sigma^{*}$. Its implementation leads to a complexity equivalent to the implementation of schedule $SW\text{-}\Sigma^{-}$ with a higher activity. The improved schedule can be applied whatever the turbo code and its interleaver. The activity comparison of schedules $SW\text{-}\Sigma^{*}$ and $SW\text{-}\Sigma^{0}$ showed that $SW\text{-}\Sigma^{*}$ is more efficient for long frame sizes or a reduced number of processors. On the contrary, for small frame sizes or a high number of processors, schedule $SW\text{-}\Sigma^{0}$ is more efficient.

In the second step, we handled the activity enhancement using the degrees of freedom available at the system level. If the latency of the decoder can be relaxed above the reception delay of a frame, we have shown that the utilization of a hybrid architecture makes it possible to decode with high activity by combining the advantages of both parallel and serial architectures: low memory requirements for large blocks through the parallel configuration, high activity for the small frame with the serial configuration.

Then, we have allowed a modification (degradation, in fact) of the iterative decoding algorithm in order to be able to use an overlapping schedule with maximal activity: the processing of the next half-iteration is started before the end of the processing of the current one. We have shown that this overlapping schedule leads to consistency conflicts defined as a tentative read access to an extrinsic sample that have yet not been computed. The proportion η of conflicts has been evaluated for a uniform interleaver: η evolves quadratically with the number of processors and the overlapping depth. The management of these conflicts during the decoding execution explains the modification of the iterative decoding algorithm: it uses a proportion η of “old” extrinsic data that have been produced at the previous iteration. In order to store the extrinsic data of the previous iteration efficiently, an architectural solution based on a hash-table has been proposed. The error rate performance evaluation of this technique has shown that a significant degradation in decoding performance spoils the throughput increase associated with this solution. However, we have also proposed two enhancements of this technique, which we believe will improve results:

- usage of a data-dependent scaling factor for scaling differently the extrinsic data used from the previous iteration.
- usage of an iteration-dependant overlapping depth that is increased during the iterations.

These two points are part of our short term perspectives.

Finally, when the designer has the freedom to design the code, and particularly the interleaver, a constrained interleaver design has been introduced. This joint code-decoder methodology is used in conjunction with an overlapping schedule in order to avoid consistency conflicts. This methodology is based on a constraint mask associated with the schedule that the interleaver must satisfy. It has been shown that even with a high number of processors, and a medium frame size, an efficient interleaver can be designed. We have shown that using a multi-level hierarchical interleaver approach associated with an MSTC, performance improvement up to either 0.2 dB in SNR or 25 % in throughput can be achieved easily. Note, however, that this solution is limited by the minimal size of the slices. To overcome this limitation, a SSTC can be associated with a hierarchical interleaver. Our first results in this direction presented in this chapter have shown that the error floor is higher than with MSTCs. Nevertheless, we believe that careful optimization of the interleaver, although time consuming, should improve this drawback.

Chapter 5

Increasing the clock frequency

Contents:

Chapter 5. Increasing the clock frequency

This chapter deals with the augmentation of the clock frequency of turbo decoder implementations. As shown in chapter 2, increasing the clock frequency is a worthwhile solution among the design space that is available to the designer. For a turbo decoder implementation, the maximal frequency corresponds to the critical path along the feedback loop in the recursion units: the so-called ACS bottleneck. Pipelining the recursion units involves a high-level modification of the decoding algorithm. This point is addressed in the first section.

After a review of the state-of-the-art techniques in the second section, we propose in the third section a technique to multiplex distinct trellis sections of the same frame on a single pipelined recursion unit. The pipeline multiplexing technique is then applied to the design of a processor for the Forward-Backward algorithm.

Finally, the last section compares the complexity and the clock frequency improvement of the processors implemented both on FPGA and ASIC circuits.

5.1 Overview

In this section, we first show evidence of the difficulty of pipelining the feedback loop of the recursion unit (RU). Then, in the second part, we give an overview of a higher level solution based on the multiplexing of two trellis sections to overcome this difficulty.

5.1.1 Pipelining the recursion unit

The maximum clock frequency of a turbo decoder implementation is generally limited by the critical path along the recursive computation of the state metrics. As described in chapter 2.2, the computation of the updated state metrics involves the classical Add-Compare-Select (ACS) operation, as depicted in Figure 5-1. The critical path therefore comprises the delay during the addition, the comparison and the selection operations. In addition, the recursion includes logic for metric rescaling and register initialization, which further increases the propagation delay. This limitation in the maximal frequency of the state metric recursion has already been studied intensively in the literature for the Viterbi algorithm. It is referred to as the ACS bottleneck.

As already mentioned in chapter 2.3, pipelining a feedback loop is not straightforward. For example, let us suppose that the feedback loop includes one intermediate register R_1 in addition to R_0, as depicted in Figure 5-2. It splits the State Metric Update (SMU) computation into two parts: the first part SMU_1 and the second part SMU_2, respectively.

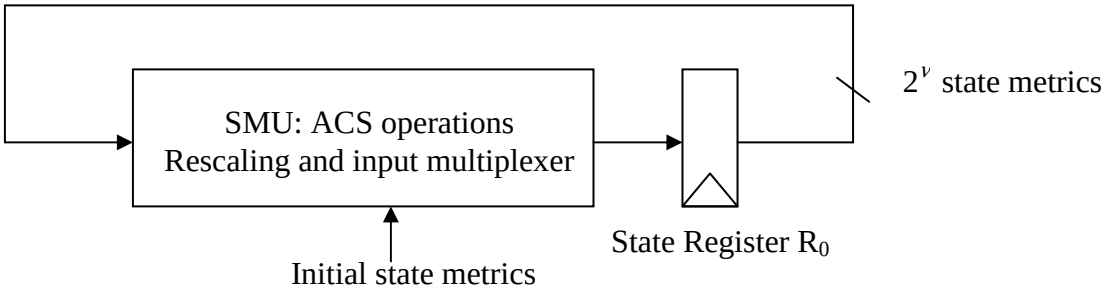


Figure 5-1: Recursion unit involving ACS operations

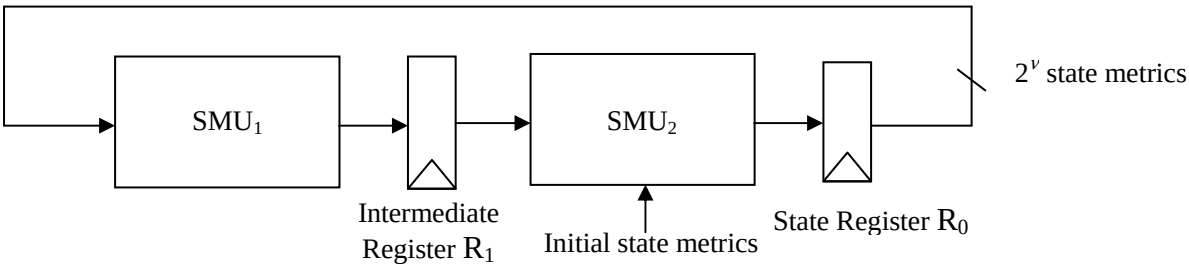


Figure 5-2: Recursion unit with an additional register inside the feedback loop

The timing behaviour of the pipelined recursion unit (PRU) is represented in Figure 5-3. Let us describe a forward recursion, for instance. Let $\mathbf{a}_k$ denote the set of forward state metrics at stage *k*, and $\tilde{\mathbf{a}}_k$ the set of intermediate results between stages *k*-1 and *k* obtained after the computation of SMU_1. At the first cycle, the first part of the SMU update (SMU_1) is computed using the current value $\mathbf{a}_k$ stored in R_0 and the result $\tilde{\mathbf{a}}_k$ is stored in R_1. At the next clock cycle, the ACS computation is completed (SMU_2) and the resulting new set of state metrics $\mathbf{a}_{k+1}$ is stored in R_0. During this cycle SMU_1 is not used since the next set of state metrics has not yet been computed completely. More generally, at each clock cycle, either SMU_1 or SMU_2 is not used: the operators in the feedback loop are thus used only half of the time, resulting in halved activity.

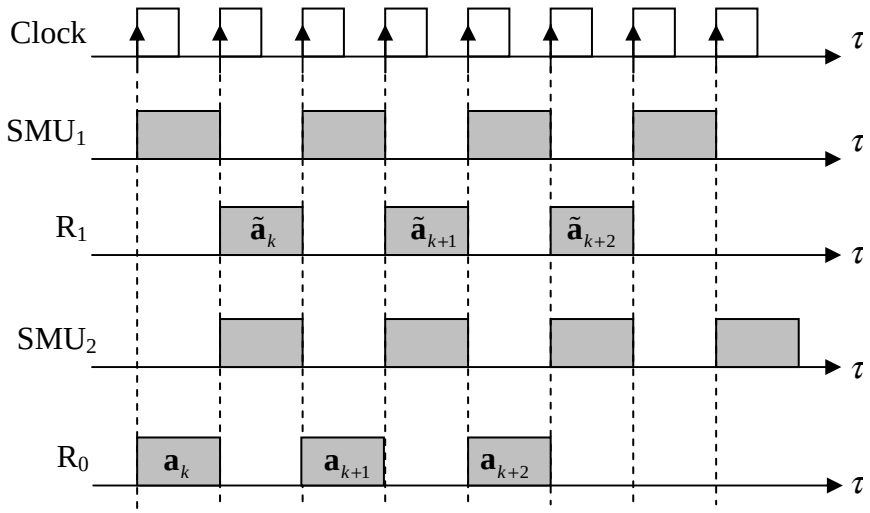


Figure 5-3: Timing of the computation of the state metrics (SMU units are active during the shaded area)

5.1.2 Pipeline multiplexing

Several techniques to introduce a pipeline register inside the recursion have been proposed in the literature, first for the implementation of the Viterbi algorithm, then for the implementation of the Forward-Backward algorithm. They rely on a pipeline multiplexing technique that simultaneously allocates the computational resources of the recursion feedback loop to distinct trellis sections. They differ in the way they exhibit the trellis sections. The principle of pipeline multiplexing is described as follows for two trellis sections.

Let us consider the RU of Figure 5-2. This PRU uses pipeline multiplexing to decode two trellis sections simultaneously. Let $\mathbf{a}_k^s$ and $\tilde{\mathbf{a}}_k^s$ denote the set of forward state metrics at stage k and intermediate results, respectively, of trellis section s (s = 1 or s = 2). During the first clock cycle (between τ_0 and τ_1), R_0 and R_1 contain $\mathbf{a}_k^1$ and $\tilde{\mathbf{a}}_{k'-1}^2$, respectively, where k and k' denote the stage indices for the first and second trellis section, respectively. SMU_1 computes $\tilde{\mathbf{a}}_{k+1}^1$ for the first section which is stored at the next rising edge τ_1 of the clock in R_1. Simultaneously, SMU_2 computes $\mathbf{a}_{k'}^2$ which updates R_0. During the next cycle (between τ_1 and τ_2), the end of the computation is completed and the result $\mathbf{a}_{k+1}^1$ is stored in R_0 at τ_2. R_1 is updated with $\tilde{\mathbf{a}}_{k'+1}^2$ resulting from the processing of SMU_1. The same process is reproduced periodically: at the even time instants, the state register contains a set of state metrics relative to the first trellis section, whereas, at the odd time instants, it contains a set relative to the second trellis section.

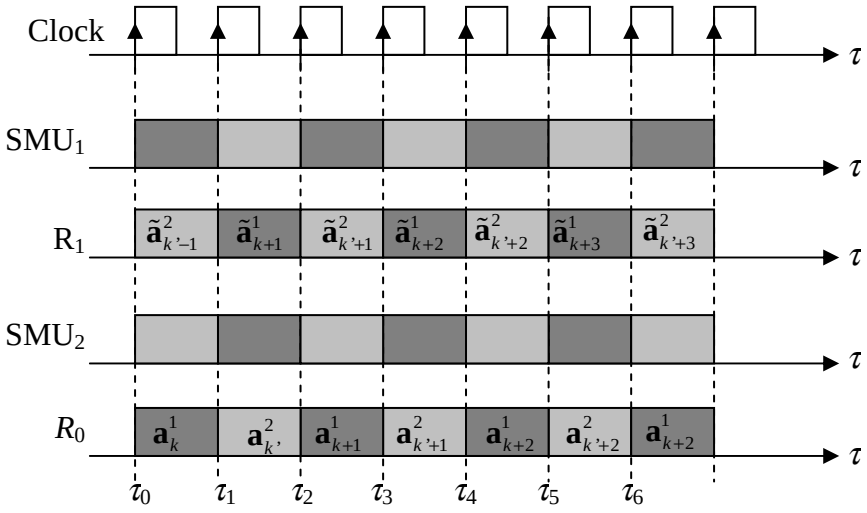


Figure 5-4: Allocation of the computation of two trellis sections using a pipeline multiplexing technique

The next section presents the state-of-the art techniques that are used to apply the above-described pipeline multiplexing method.

5.2 State-of-the-art

This second section describes the state-of-the-art techniques that have been proposed in the literature to exhibit distinct trellis sections (and thus recursions) that can be multiplexed on the PRU introduced in the previous section.

5.2.1 Multiplexing independent streams [Lin *et al*.-89]

In [Lin *et al*.-89], Lin and Messerschmitt propose two pipeline multiplexing techniques based on the decoding of independent convolutional-encoded streams.

The first technique described in Figure 5-5 involves the transmission of independent sources of information. The sources are encoded separately by an identical convolutional encoder. The resulting independent streams are then transmitted through a time-division multiple-access channel. Then, at the decoder side, a single Viterbi decoder processes the independent streams concurrently. Using this method, the throughput of the transmission of each source is not increased, but only the throughput of the decoder, which is in this case efficient.

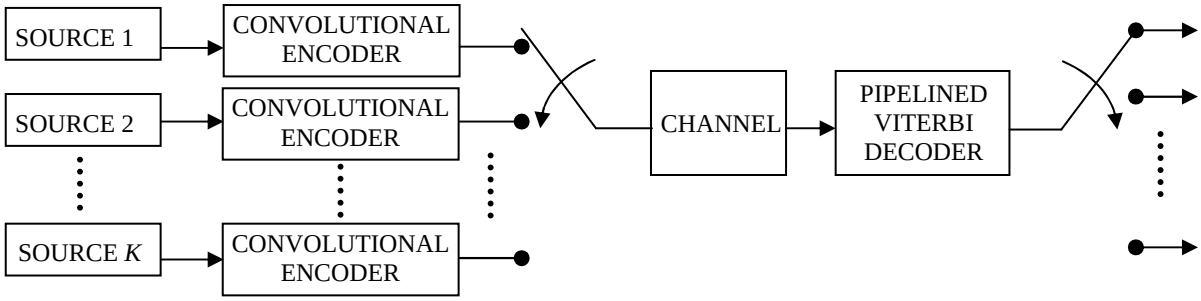


Figure 5-5: Concurrent Viterbi decoding of independent sources

The second technique uses a single source, whose stream is multiplexed on several independent convolutional encoders, as depicted in Figure 5-6. Similarly, the concurrent Viterbi decoder processes the independent and infinite streams simultaneously. The decoded streams are then merged together in order to recover the information stream.

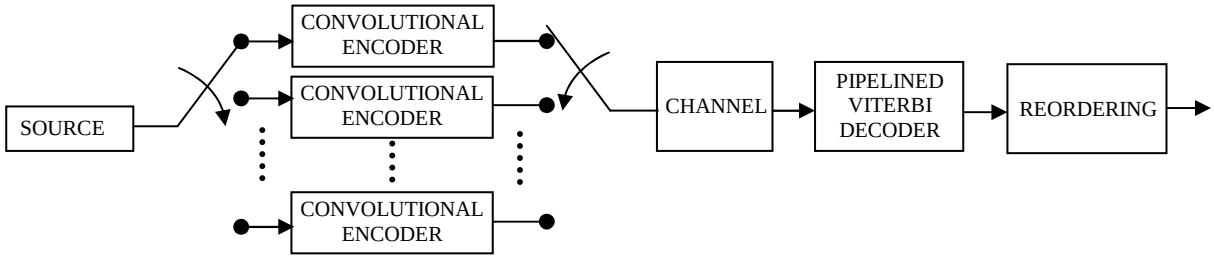


Figure 5-6: Concurrent Viterbi decoding of independent encoded streams

Unlike the preceding technique, this latter enables high-throughput decoding. However, it is restrictive, since it imposes a modification on the encoding scheme. The next section will describe a technique suitable for terminated frame transmission.

5.2.2 Multiplexing successive frames [Lin *et al.*-89]

This idea of multiplexing successive frames was first introduced in [Lin *et al.*-89] using the terminated Viterbi encoding of a single source. The formation of several trellis sections is performed by periodically terminating the convolutional code using either tail-bits or forcing the encoder state to the "all-zero" state. The successive trellis sections are then decoded by a concurrent Viterbi decoder (see Figure 5-7). The application of this idea to convolutional encoding is not very efficient, since it imposes a modification on the encoding process, which either degrades the performance (zero-forcing) or reduces the code rate (tail-bits).

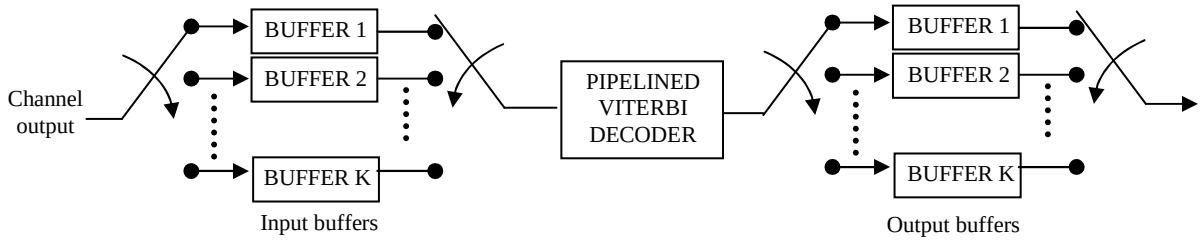


Figure 5-7: Multiplexing successive frames

The same idea could be applied successfully to any terminated turbo code, using a single decoder that decodes several successive frames concurrently. For unmodified frame size, it comes at the expense of a large memory requirement because the data relative to the frames decoded concurrently must be stored. The memory requirements could remain unchanged by dividing the frame size by K but would degrade the performance of the turbo code significantly [Dolinar *et al.*-98].

The next sections present techniques exhibiting trellis sections of a single encoded frame.

5.2.3 "Horizontal" trellis partition [Dawid *et al.*-92]

The authors of [Dawid *et al.*-92] propose a technique to split the 2^{ν} nodes of one trellis stage into two separate groups of $2^{\nu-1}$ nodes. This partition is performed by noting that for a convolutional code, at a given trellis stage, there exist two groups of nodes, whose branches are linked to common nodes of the next trellis stage. These common nodes are distinct for the two initial groups of nodes and therefore form two independent groups again. As shown in Figure 5-8, the two groups issue two independent sub-trellises that remain independent for U-1 trellis stages, where U is a trellis-dependent constant ($U = 4$ in our example). At the U-th trellis stage, the sub-trellises merge and become independent again, from the next stage on. This separation of the nodes of the trellis into two groups explains the term "horizontal" partition.

This trellis partition exhibits independent sub-trellises that can be decoded using a pipeline multiplexing technique with $2^{\nu-1}$ pipelined ACS operators. The two parts of the pipelined ACS operators are allocated simultaneously to the two sub-trellises. The allocation is inverted at every clock cycle. Nevertheless, one waiting cycle is inserted periodically, when the two sub-trellises converge (*i.e.* every U trellis stages). These waiting cycles lead to a reduction in the activity of the recursions expressed as $2 \cdot U / (2 \cdot U + 2)$. Since the value of U is trellis-dependent,

and in particular depends on the number of states and branches of the trellis, this technique is efficient only for a trellis with a large number of states and a reduced number of transitions per state, so that U is high. This is completely irrelevant for the trellis of the double-binary turbo code (3,2,3) with 8 nodes per trellis stage and 4 branches per node, for which $U = 2$.

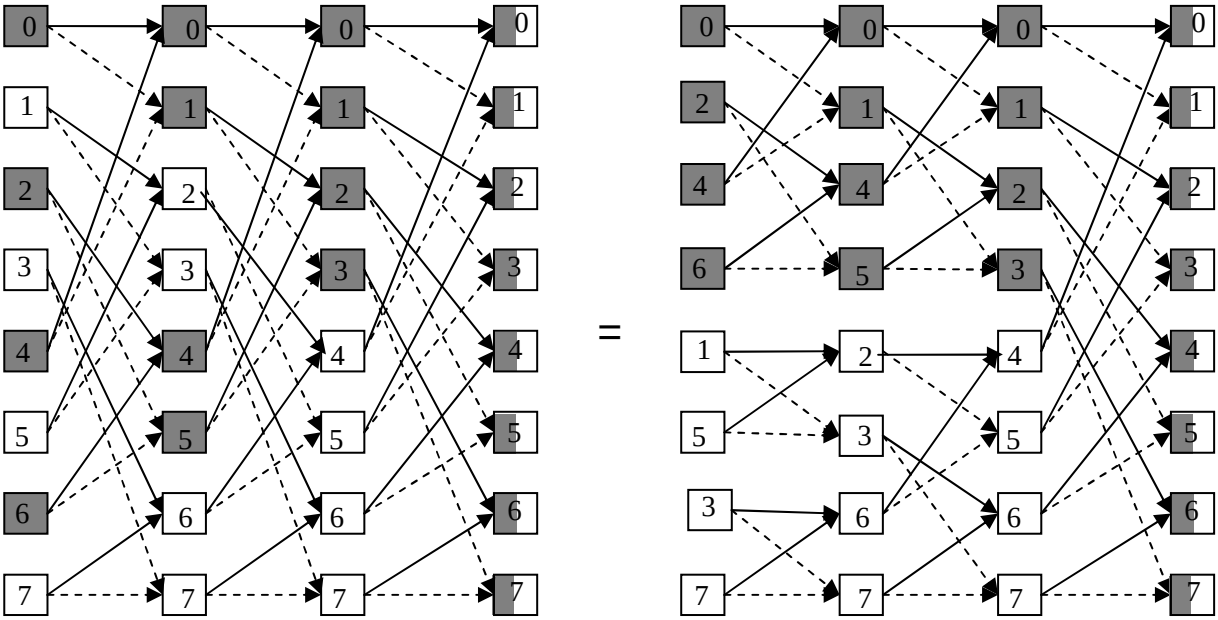


Figure 5-8: "Horizontal" splitting of the recursions into two sub-trellises

5.2.4 Multiple trellis stage decoding [Fettweis *et al*.-89]

Another technique developed in [Fettweis *et al*.-89] for the Viterbi algorithm introduces the concept of the M-step trellis: the Viterbi recursion performs M trellis stages simultaneously. Since M trellis stages are computed simultaneously, there exist several branches of the M-step trellis between a starting and an ending node (see Figure 5-9). This multiplicity of branches leads to the computation of as many branch metrics, and in particular, the selection of the minimal branch metric. Clearly, all paths starting from the same state and ending at the same state have the same initial state metric. As a consequence, the selected path among this set of paths corresponds to the path with the minimal branch metric.

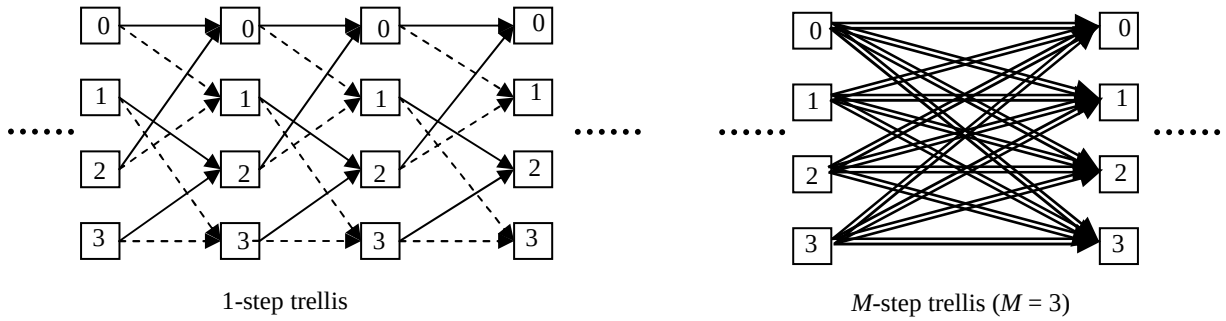


Figure 5-9: One-step trellis and corresponding M-step trellis ($M = 3$)

The authors have shown in [Fettweis *et al*.-89] that the computation of the minimal branch metric for each set can be performed using a Viterbi algorithm on a dedicated sub-trellis.

Thus, the Viterbi algorithms performed in parallel on distinct sub-trellises can be multiplexed on a single PRU according to the pipeline multiplexing technique.

Unfortunately, the technique of the *M*-step trellis is not adapted to the Forward-Backward algorithm since this latter requires the state metrics at each trellis stage in order to compute its soft outputs.

5.2.5 Multiplexing independent recursions of the same trellis [Park *et al*.-00]

The authors of [Park *et al*.-00] apply the pipeline multiplexing technique to the decoding of a single binary convolutional code decoded using the LogMAP algorithm. The schedule *SW-*Σ^- is used in conjunction with pre-processing steps to initialize the backward recursions. This schedule depicted in Figure 5-10 uses three RUs: one forward and two backward RUs.

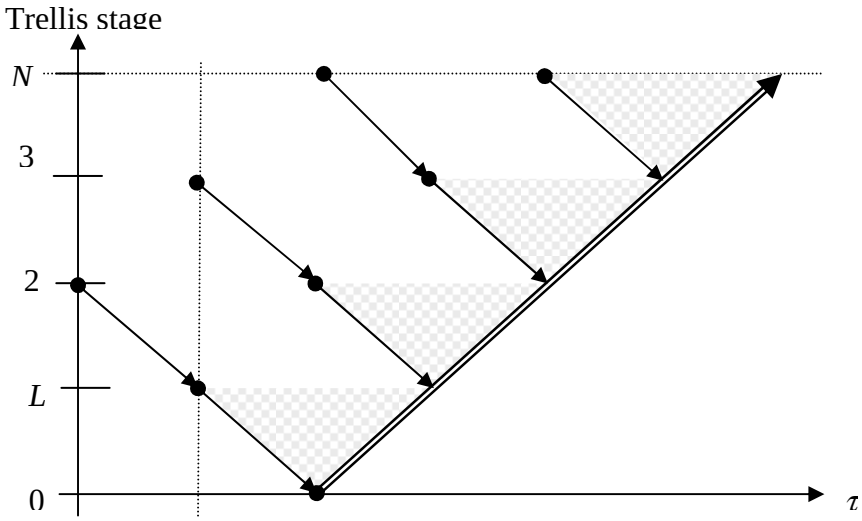


Figure 5-10: Schedule *SW-*Σ^- with pre-processing steps to initialize the backward recursions

Since the optimal Log-MAP algorithm is performed, each RU contains an ACS operation followed by the addition of a corrective term extracted from a Look-Up-Table (LUT). This RU has been separated into three parts by adding two additional pipeline registers as represented in Figure 5-11. Then, the three independent recursions of Figure 5-10 are processed on this single PRU. However, it comprises a programmable shuffle network inside the feedback loop to be able to process in the forward and in the backward directions, since the shuffle network is different for the two directions as seen in chapter 2.2. The programmability capability of the shuffle network requires multiplexers that increase both the complexity and the propagation delay along the feedback loop. For this reason, this technique lessens the expected augmentation of the clock frequency.

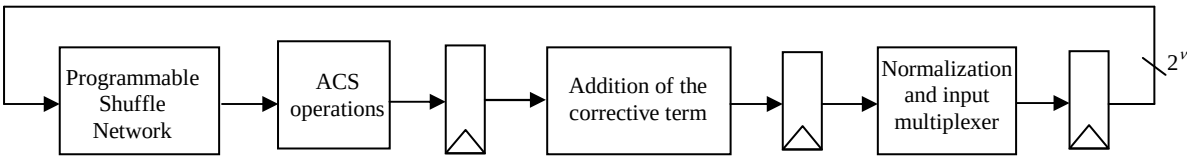


Figure 5-11: Pipelining a recursion unit using the *max operator for a binary convolutional code**

5.2.6 Discussion

The techniques described in this section cannot be used for turbo-decoding convolutional codes, and especially double-binary turbo codes. They require either a modification of the encoding principle (5.2.1) or are simply not suited to the Max-Log-MAP algorithm (5.2.4). For the techniques that are suited to the Max-Log-MAP algorithm and to the decoding of turbo codes we have shown that they are not efficient enough. They show either a low activity for double-binary RSC codes (5.2.3), an increase in complexity (5.2.5) or an augmentation in memory requirement (5.2.2).

In the next section, we propose another technique to exhibit distinct trellis sections used to decode a single turbo-coded frame efficiently.

5.3 Pipeline multiplexing of trellis sections for turbo decoding

In this section, we propose an efficient pipeline multiplexing technique for the Forward-Backward algorithm. In the first step, we describe our solution to exhibit the distinct trellis sections. Then, after a brief study of the propagation delay along an ACS operator, the outcomes are used to pipeline the RUs appropriately. Finally, the PRUs are integrated in the processors designed for the two schedules *SW*-Σ^{-} and *SW*-Σ^{*}.

5.3.1 Overview

The technique we propose here in order to decode a single trellis with the Forward-Backward algorithm is to process several similar recursions (forward or backward) on a dedicated RU (forward or backward, respectively). The recursions that are processed on the same RU belong to distinct trellis sections. They correspond to the ones that have been devised for the parallel architecture, as detailed in chapter 2.3. We explained that for a parallel architecture, the trellis sections are decoded simultaneously by dedicated processors. Each processor comprises a forward and a backward RU. Using pipeline multiplexing, several - the number depends on the number of pipeline stages - similar RUs of distinct processors are replaced by a single PRU. Pipelining the other computational units (*e.g.* branch metric computation and soft output computation units), thus working at the same clock frequency, enables us to share them between the trellis sections that are processed by the PRUs. This can be carried out easily since the other computation units do not include a feedback loop.

Consequently, introducing J pipeline stages inside each of the RUs makes it possible to replace J processors by a single pipelined processor. Nonetheless, introduction of pipeline registers has a cost in complexity, since a single pipelined processor requires more registers than a single non-pipelined processor. Fortunately, for FPGAs there is no complexity increase because the additional registers that are needed are already available. It was shown in chapter 2.1, that each logic element (LE) is linked to a register. When the critical path involves several stages of LEs, the registers associated with the LEs, except the last one, are not used[1]. As a result, introducing additional registers to increase the clock frequency is done at no

[1] They can in principle be used by some other function, but, in practice, this is rarely the case.

additional cost in complexity. This will be verified in section 5.4.1, and the additional complexity will also be evaluated for an ASIC implementation.

Before discussing the insertion of a pipeline register into the RU, we give a brief description of the propagation delay along an ACS operator.

5.3.2 Propagation delay along an ACS operator

Let us start with a conventional implementation of two successive additions. The critical path includes propagation through the two adders successively, as depicted in Figure 5-12. An adder of *b* bits is implemented by a succession of *b* full adders (FA). Each FA is associated with one bit of each operand. This provides the corresponding bit of the result, and the carry, which is propagated along all the FAs. Hence, the critical path in an adder corresponds to the propagation of the carry from the Least Significant Bit (LSB) to the Most Significant Bit (MSB):

$$t_{ADDER} = b \cdot t_{FA}, \qquad \textbf{(5-1)}$$

where t_{ADDER} corresponds to the propagation delay of the adder, and t_{FA} corresponds to the propagation delay through the FA. Equivalent propagation delays are assumed for the paths from all inputs to every output of the FA.

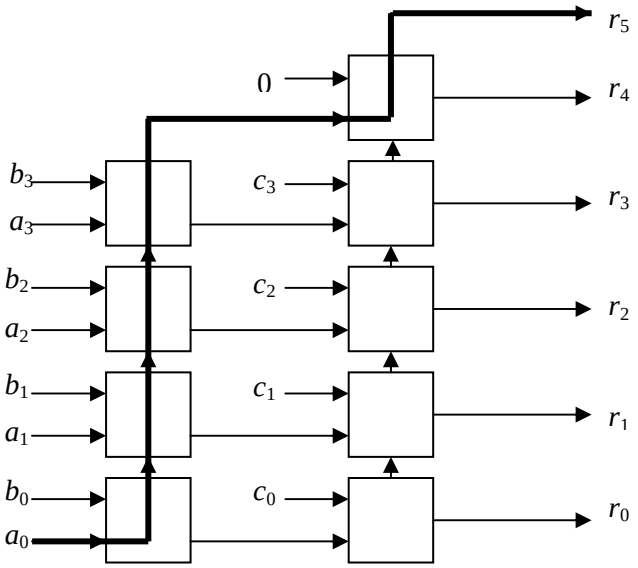


Figure 5-12: Critical path for two successive additions *r* = *a* + *b* + *c*

However, the critical path along the two successive adders is not twice the propagation delay of a single adder, as once would expect (see Figure 5-12). This is explained by the fact that the computation of the LSBs for the second adder can start before the end of the computation of the MSBs for the first adder. As a consequence, the propagation delay through two adders equals

$$t_{ADDER} = t_{ADDER} + t_{FA}. \qquad \textbf{(5-2)}$$

This rule naturally extends to a succession of more than two adders or subtracters. It is well known that for a subtracter, using the 2-complement representation, the subtraction of two

operands results in an addition. The operand that is subtracted is replaced by the addition of the inverse operand and an additional input carry for the FA of the LSB:

$$a - b = a + (-b) = a + \overline{b} + 1, \qquad \textbf{(5-3)}$$

where $\overline{b}$ is obtained by the inversion of all bits of *b*.

For a conventional implementation of an ACS operation on an FPGA, the comparison-selection operation is necessarily implemented as a subtraction-selection operation, which is composed of two LEs per bit, one for the subtraction and the other for the selection. Hence, the propagation delay through the adder and the comparator follows the same above-mentioned rule for two adders. This observation is taken into account in the next sub-section to insert a pipeline register appropriately.

5.3.3 Pipelining the state metric recursion unit

After outlining the inefficiency of pipelining an RU computing the Max-Log-MAP algorithm on a binary trellis (with two branches per state), we discuss the insertion of an additional register into an RU performing this algorithm on a double-binary trellis (with 4 branches per state). Then, the same work will be done for an RU performing the Log-MAP algorithm (with a corrective factor as detailed in chapter 1.5) on a binary trellis.

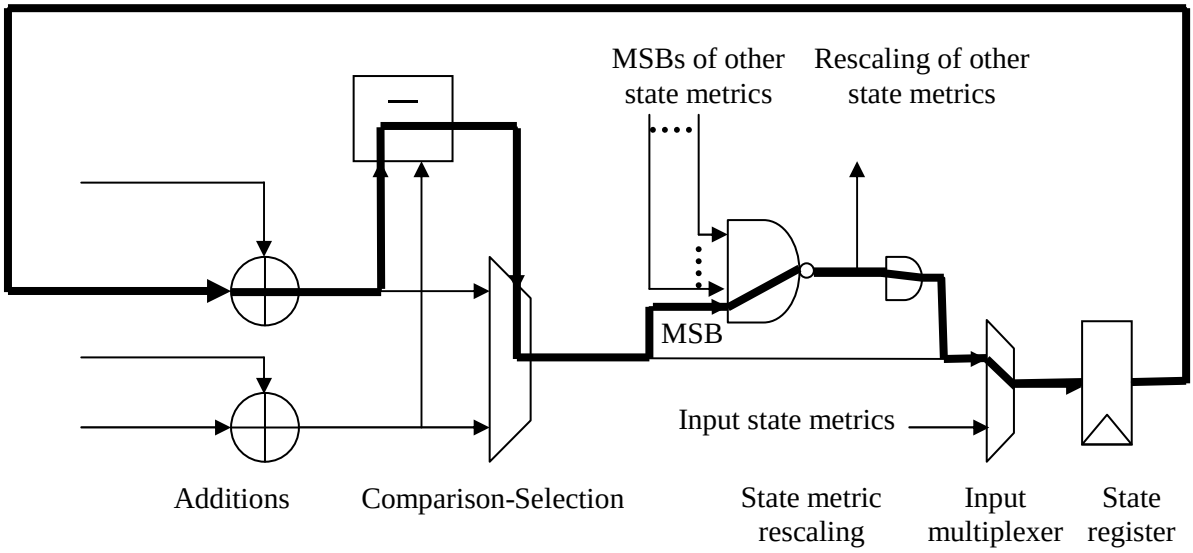


Figure 5-13: Critical path of the feedback loop for a binary trellis

Let us first consider an RU processing a Max-Log-MAP algorithm for a binary trellis. The corresponding feedback loop includes an addition, a comparison-selection operation, the rescaling operation and a multiplexer used to initialize the recursion with an initial set of state metrics. Therefore, as shown in Figure 5-13, the critical path corresponding to the update of a state metric includes the propagation through an adder, a subtracter, a multiplexer for the selection, two logical operators (NAND and AND for rescaling) and another multiplexer. Inserting a pipeline register inside the feed-back loop can be done easily between the subtracter and the multiplexer. But this solution does not lead to two equivalent propagation delays between the two registers. To this end, the register needs to be inserted inside the

addition-comparison operation. Unfortunately, pipelining the RU for a binary trellis does not necessarily lead to a multiplication by two of the clock frequency of the whole circuit. The reason is that a multiplication of the frequency by two in the other computational units of the decoder cannot always be reached since the clock frequency is already high and approaches the limits of the circuit. This is especially true for FPGA circuits. For this reason, pipeline multiplexing has not been used so far for the implementation of binary turbo codes decoded with the Max-Log-MAP algorithm.

On the other hand, for a double-binary RSC code the conclusion is not the same. The ACS operation for a double-binary trellis corresponds actually to 4 additions and a two-level tree of comparison-selection operations. As shown in Figure 5-14, the critical path corresponding to the update of a state metric includes an additional set of subtracter-multiplexer. Obviously, the propagation delay is longer than for a binary trellis, which contains only one stage of comparison-selection operation. It is thus worth pipelining the RU in order to increase the clock frequency of the circuit.

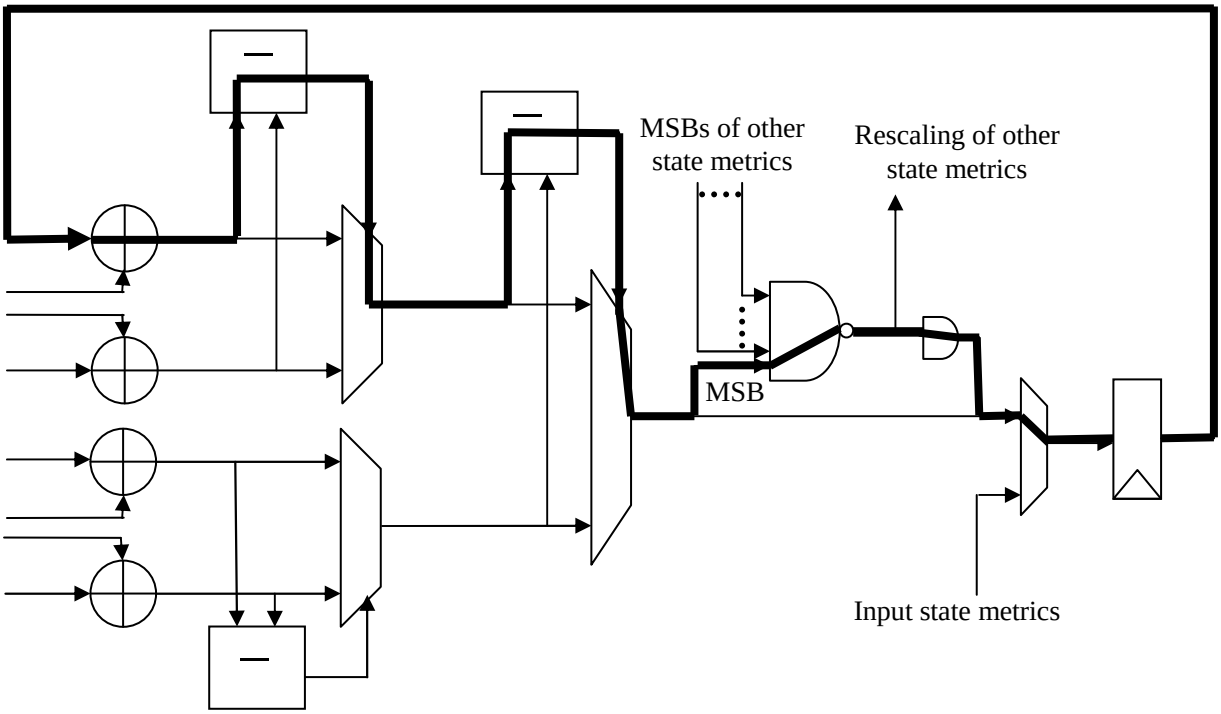


Figure 5-14: Critical path of the feedback loop for a double-binary trellis

Taking into account the observations made earlier about propagation through the succession of an adder and a subtracter, the additional pipeline register is inserted between the two stages of the comparison-selection tree as depicted in Figure 5-15. It has been verified by synthesis that the paths between the two registers are equivalent. The synthesis results for a Virtex II FPGA of Xilinx are given in Table 5-1.

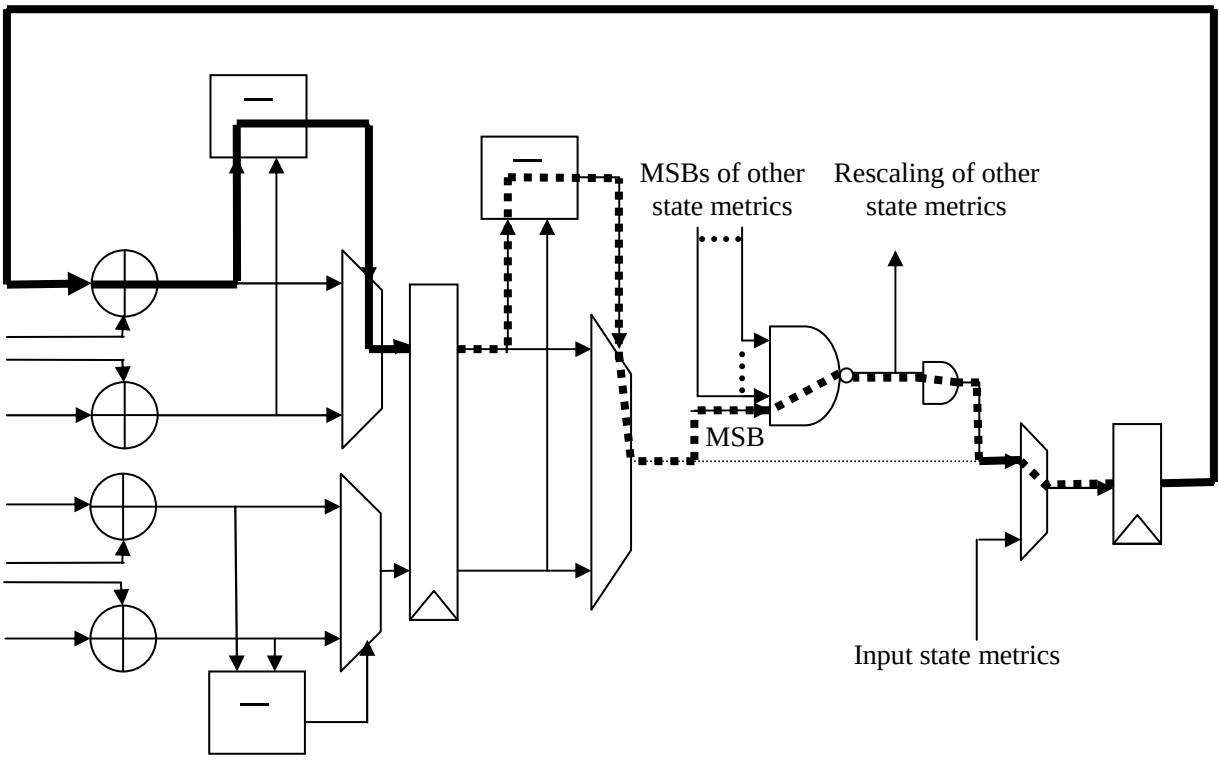


Figure 5-15: Pipelined recursion unit for a double-binary trellis

		Minimal propagation delay (ns)	Maximal frequency (MHz)
Non-pipelined recursion unit		12.494	80
Pipelined recursion unit	**state register → intermediate register**	6.647	150
	intermediate register → state register	6.591	

Table 5-1: Minimal propagation delay for a Virtex II FPGA for the paths inside the recursion unit processing the Max-Log-MAP algorithm on a double-binary trellis

Finally, pipelining the RU for a binary RSC code decoded using the Log-MAP algorithm is also worthwhile, since the propagation delay includes an additional adder for the addition of the corrective term. The register can be inserted successfully between the comparison-selection operator and the adder as shown in Figure 5-16. The synthesis results for a Virtex II FPGA are given in Table 5-2.

		Minimal propagation delay (ns)	Maximal frequency (MHz)
Non-pipelined recursion unit		15.337	65
Pipelined recursion unit	**state register → intermediate register**	8.203	122
	intermediate register → state register	8.104	

Table 5-2: Minimal propagation delay for a Virtex II FPGA for the paths inside the recursion unit processing the Log-MAP algorithm on a binary trellis

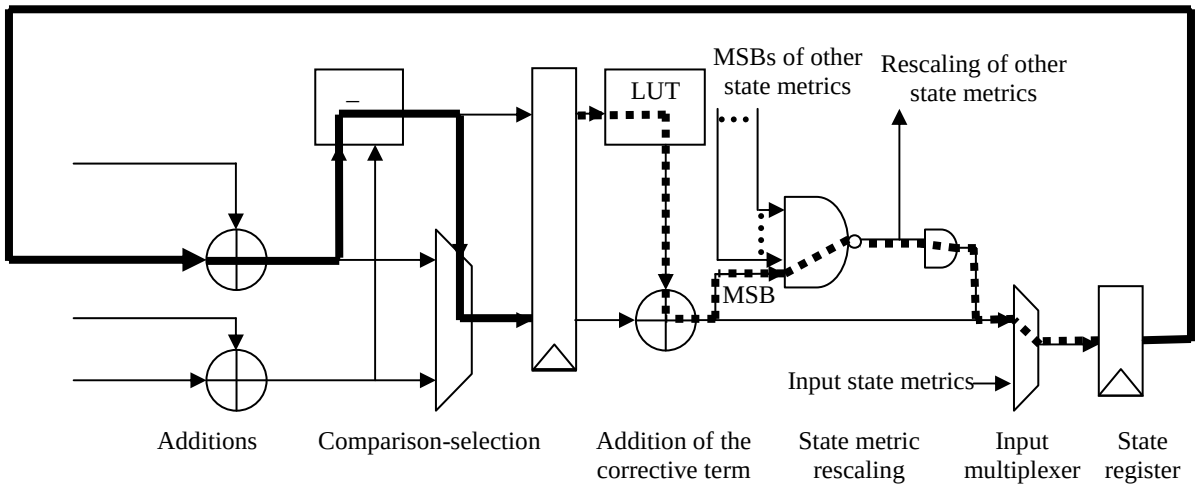


Figure 5-16: Pipelined recursion unit for a binary trellis decoded with the Log-MAP algorithm

5.3.4 Application to the Forward-Backward algorithm with schedule *SW-Σ⁻*

Using the PRU described in the previous section, a pipelined processor performing the *SW-Σ⁻* schedule can be designed. The other computation units are also pipelined so that the critical path of the architecture remains inside the feedback loop of the RUs. The total number of latency cycles due to the pipeline is thus approximately doubled to $l_p = 16 = 2 \cdot l$ compared to a non-pipelined processor of latency $l = 8$. This number has been verified by the design and synthesis of the whole pipelined processor on FPGA and ASIC circuits (see section 5.4.2).

The pipelined processor exhibits two virtual processors, performing the same *SW-Σ⁻* schedule depicted in Figure 5-17 on distinct trellis sections. The corresponding architecture and allocation of the resources to the decoding of the two trellis sections are also represented in this figure. The architecture comprises the following modules:

- two pipelined branch metric computation units PBMU F and PBMU B,
- one pipelined backward recursion unit PBRU,
- one pipelined forward recursion unit PFRU,
- one pipelined soft output computation unit PSOU.

The architecture further comprises two memories of size $2L$ words storing the data relative to the two trellis sections:

- the Extrinsic and Observation Sample memory EOS MEM,
- the backward state metric memory BSM MEM.

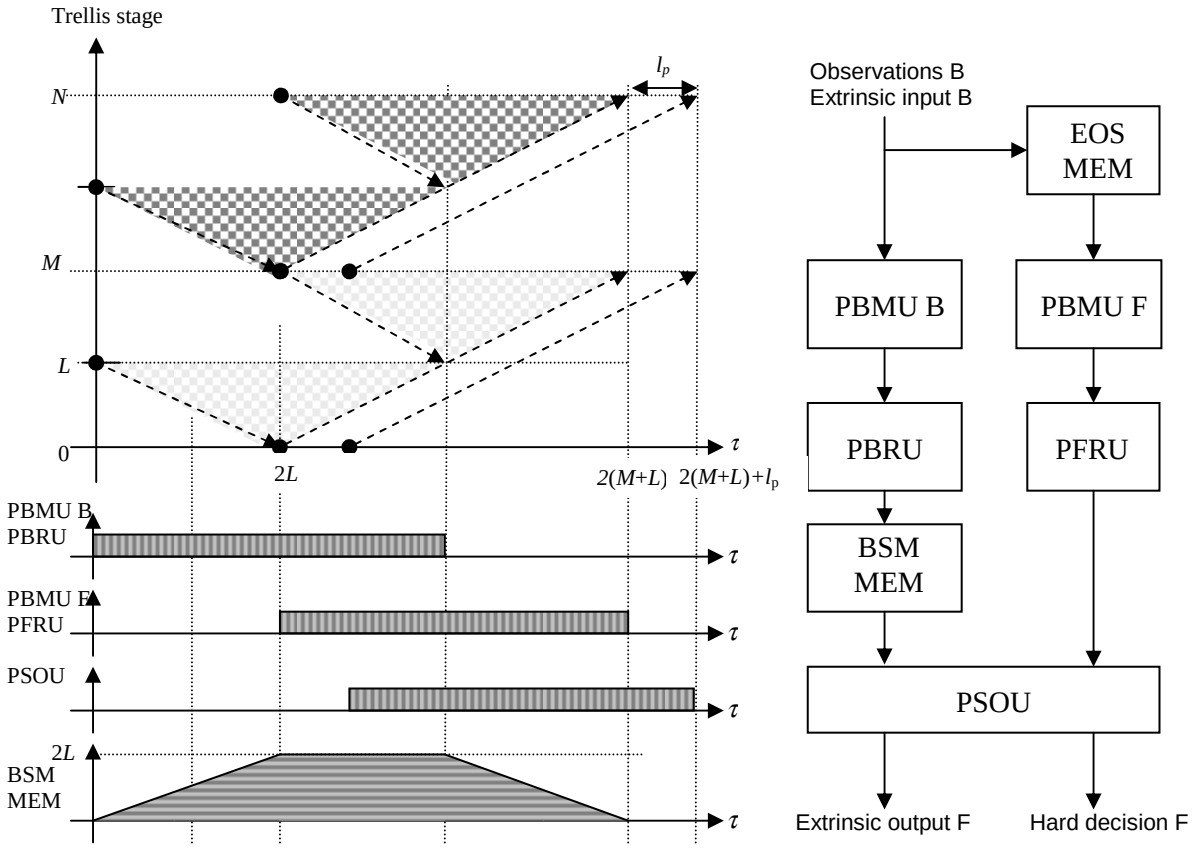


Figure 5-17: Pipelined processor performing schedule *SW-Σ⁻* and allocation of the computation to the resources

The activity of the pipelined processor is expressed as

$$\alpha^{p}_{SW-\Sigma^{-}} = \frac{2 \cdot M}{2 \cdot (M + L + l)} = \frac{N}{N + 2 \cdot (L + l)}. \qquad \textbf{(5-4)}$$

It is worth noting that compared to a non pipelined processor, the fixed overhead is doubled, resulting in a reduction in activity. In fact, considering the two virtual processors, the activity of the architecture with the pipelined processor equals the activity of a parallel architecture with two processors.

This architecture can be generalized to P pipelined processors, resulting in $2 \cdot P$ virtual processors processing in parallel the same schedule *SW-Σ⁻* with activity

$$\alpha_{SW-\Sigma^{-}} = \frac{N / 2P}{N / 2P + L + l}. \qquad \textbf{(5-5)}$$

5.3.5 Application to the Forward-Backward algorithm with schedule *SW-Σ**

Similarly, a pipelined processor can be designed for schedule *SW-Σ**. The design of such a processor starts from the architecture of the two processors given in chapter 4.1. Then, the two forward recursion units FRU_1 and FRU_2 are merged into a single pipelined forward recursion unit PFRU. In a similar manner, the backward RUs are merged, and the same applies for the branch metric computation units and soft output computation units. Of course,

multiplexers are needed before and after the PRUs. The architecture and the corresponding allocation of the computations are represented in Figure 5-18. PFRU is used to process the forward recursion of the first and the second trellis section, alternately. In the same way, PBRU is used to process the two trellis sections alternately. In contrast, the utilization of PSOU requires a double alternation. From time $\tau = L$ to $2L$, it computes the soft outputs in the forward and backward directions, alternately, of the first trellis section. Then, from time $\tau = 2L$ to $3L$, it computes the soft outputs in the forward and backward directions, alternately, of the second trellis section. In summary, PSOU alternates first on the basis of a half-window processing between one trellis section and the other. Second, within the same half-window processing, it alternates at each clock cycle in order to compute the soft outputs in the forward and backward directions.

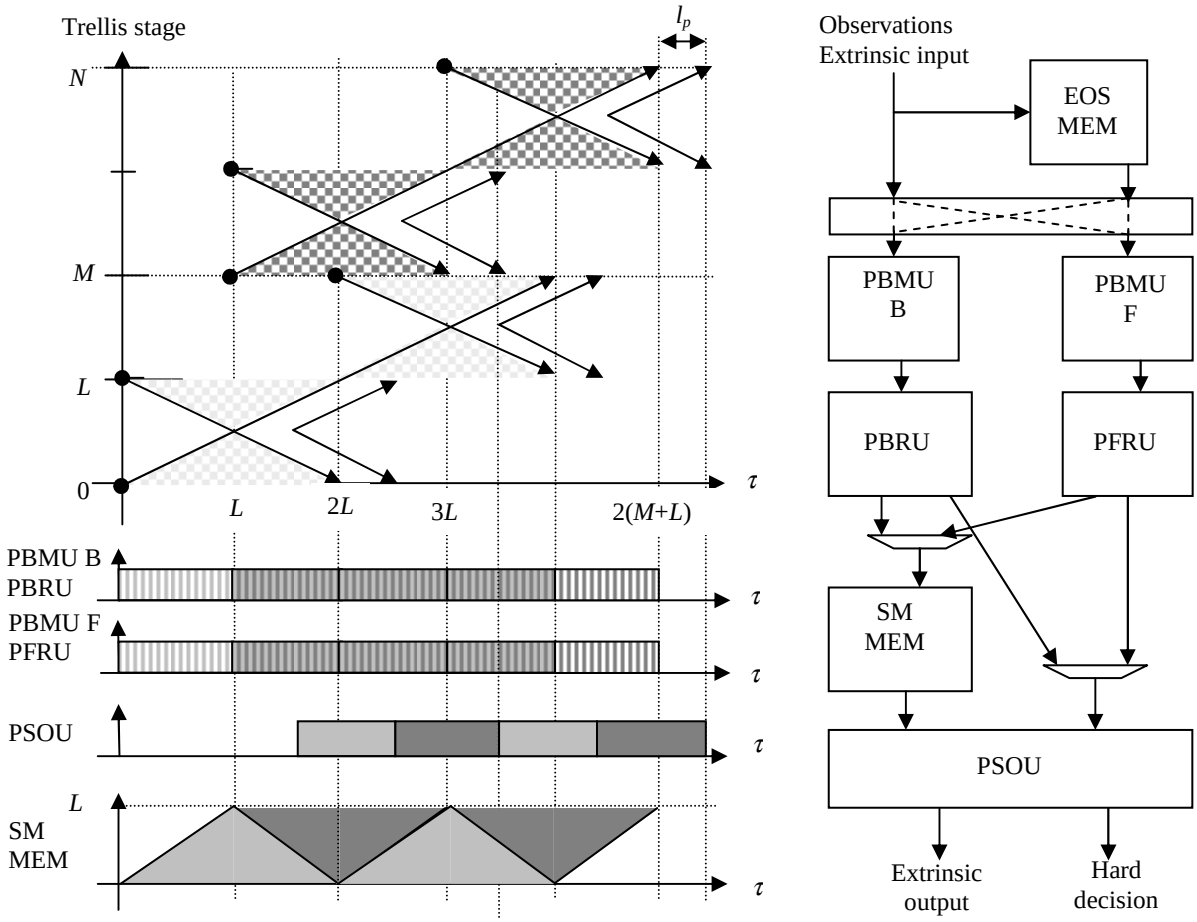


Figure 5-18: Pipelined processor performing schedule *SW-Σ and allocation of the computation to the resources**

In the same way as for schedule *SW-Σ*$^{-}$, the architecture can be generalized to P pipelined processors, resulting in $2 \cdot P$ virtual processors processing in parallel schedule *SW-Σ** with activity

$$\alpha_{SW-\Sigma^*} = \frac{N/2P}{N/2P + L/2 + l}. \qquad \textbf{(5-6)}$$

5.4 Complexity comparison

In this section, the complexities of pipelined and non-pipelined processors, given after synthesis, are compared first for FPGA circuits, then for ASIC circuits.

5.4.1 FPGA circuit

The processor implementing schedule *SW-Σ*$^{-}$ for a double-binary RSC code (8,2,4) has been synthesized using the technique of pipeline multiplexing. The quantization of input data was set to $w_d = 4$. The results obtained with the Synplify Pro 7.4 from Synplicity are given in Table 5-3 for a Xilinx Virtex II-2000 circuit using the fastest speed grade (6). The complexity and maximal frequency are compared for a standard non-pipelined processor and a pipelined processor. In the two cases, the critical path of the processor corresponds to a path inside the feed-back loop of the RUs.

	Complexity (LE)	Frequency (MHz)	Pipeline depth
Standard (non-pipelined processor)	3227	75	$I = 8$
Pipelined processor	3214	145	$I_p = 16$

Table 5-3: Synthesis results of the pipelined processor with $w_d = 4$ for Xilinx Virtex II

The results clearly demonstrate that for identical complexity, the clock frequency has been almost doubled with a pipelined processor, at no cost in complexity. This validates the assumption that the registers used for pipelining do not lead to a complexity penalty. One can also note that the depth of the pipeline is doubled, as expected.

5.4.2 ASIC circuit

The processor implementing schedule *SW-Σ*$^{-}$ for a double-binary RSC code (8,2,4) was synthesized using the 0.18µ standard library provided by ST Microelectronics. The synthesis tool Design Compiler (DC) from Synopsys was used. The results are summarized in Table 5-4. Contrary to the FPGA circuit, the doubling of the clock frequency is accompanied here with a complexity augmentation of around 25 %. Nevertheless, the technique of pipelined multiplexing is still efficient, since doubling the number of processors would lead to an augmentation of 100 % in complexity.

	Complexity (gates)	Frequency (MHz)	Pipeline length
Standard (non-pipelined processor)	15398	200	8
Pipelined processor	19231	400	16

Table 5-4: Synthesis results of the pipelined processor with $w_d = 4$ for a ST 0.18µ standard library

5.5 Conclusion

In this chapter, we have addressed the problem of limitation of the maximal clock frequency of the circuit due to the so-called ACS bottleneck. We have proposed a solution for increasing the clock frequency of the SISO processor.

We have demonstrated that a pipeline multiplexing technique can be applied successfully to decode turbo codes. This technique is based on the tiling of the trellis into several trellis sections introduced for parallel decoding architectures. Using pipelined recursion units, several trellis sections are decoded simultaneously in a time division manner on a single pipelined recursion unit. The application to the design of a processor implementing the Max-Log-MAP algorithm has been performed for the two schedules *SW-*Σ^{-} and *SW-*Σ^{*}. The complexity comparison conducted for FPGA and ASIC circuits, demonstrates the relevance of the technique. On both targets, a doubling of the clock frequency has been achieved at no additional cost for FPGA circuits, while for ASIC circuits it is associated with a complexity increase of 25 %. However, the doubling of the clock frequency does not induce an equivalent throughput augmentation. In fact, the activity is reduced similarly to that of a parallel architecture with twice the number of processors.

In the next chapter, we will use our pipeline multiplexing technique in order to design a multi-processor decoder for double-binary turbo codes.

Chapter 6

Application: Design of a multi-processor turbo decoder for WiMAX

Contents:

Chapter 6. Application: Design of a multi-processor turbo decoder for WiMAX

This final chapter describes an application of the techniques developed in the previous chapters to design a turbo decoder for the WiMAX standard (IEEE 802.16) [WiMAX]. This open specification contains a Parallel Convolutional Code option (a turbo code), using an ARP interleaver that in chapter 3.7 was shown to be suitable for parallel decoding. Our study will also use the results obtained in chapter 4 to reach high activity with parallel decoding. Since the turbo code is double-binary, the multiplexing technique proposed in chapter 5 can be applied to our decoder design.

After a brief description of the WiMAX turbo code in the first section, the second section describes the implementation of the turbo decoder. This description includes the discussion of the architectural choices that have been made and the low-level optimization of the hardware implementation of the processor. Our decoder is scalable in terms of number of input bits and number of processors, and handles a wide range of frame sizes.

Then, the third section gives the synthesis results (complexity and clock frequency) of the complete turbo decoder implementation. This latter was designed for different FPGA circuits.

Finally, in the fourth section, a performance study is carried out, in order to evaluate the impact of the quantization and of the number of processors on the error decoding performance. Then, taking into account complexity, we investigate the efficiency of the architecture as a function of the input quantization of the decoder. This work leads us to an optimal choice of input quantization, whose performance is simulated for the whole range of frame sizes.

6.1 Description of the WiMAX turbo code

The IEEE 802.16 WiMAX is a standard for fixed Broadband Wireless Access (BWA), which is dedicated to building Metropolitean Area Networks (MANs) supporting applications such as Internet access and telephony.

6.1.1 Overview

The standard defines various physical layers (PHY). In this chapter, we consider only the PHY layer based on Orthogonal Frequency Division Multiplexing (OFDM) using frequency bands below 11 GHz. This PHY layer includes several options for the forward error correcting function, one of them being a parallel concatenated double-binary convolutional turbo code. It also uses various modulation schemes (QPSK, 16QAM, 64QAM, etc.) but, in our study, we only deal with the QPSK modulation. In this configuration, the information data throughput at the decoder output can be up to 75 Mbit/s.

6.1.2 Convolutional turbo code description

The parallel concatenated convolutional turbo code of rate 1/3 uses two double-binary CRSC codes of parameters (3, 2, 4), described in chapter 1.2. The frame size in number of duo-binary symbols is denoted N. The turbo code is associated with an ARP interleaver whose equation is given by $\Pi(i) = \lambda \cdot i + \mu(i \bmod 4) + 1$, where λ is chosen to be relatively prime with N, $\mu(0) = 0$, $\mu(1) = N/2+P_1$, $\mu(2) = P_2$ and $\mu(3) = N/2+P_3$. This turbo code supports a wide range of frame sizes with a byte granularity ranging from $N = 24$ to $N = 2400$ double-binary symbols. The frame sizes and the associated interleaver parameters are given in Table 6-1. The same turbo code is punctured to generate code rates 1/2, 3/4 and 5/6, in addition to the primitive code rate of 1/3.

Frame size N	λ	P_1	P_2	P_3
24	5	0	0	0
36	11	18	0	18
48	13	24	0	24
72	11	6	0	6
96	7	48	24	72
108	11	54	56	2
120	13	60	0	60
144	17	74	72	2
180	11	90	0	90
192	11	96	48	144
216	13	108	0	108
240	13	120	60	180
480	13	240	120	360
960	13	480	240	720
1440	17	720	360	540
1920	17	960	480	1440
2400	17	1200	600	1800

Table 6-1: Frame sizes and corresponding interleaver parameters

The throughput of 75 Mbit/s required for the turbo decoder corresponding to this turbo code induces the need for a multi-processor architecture. This approach is made possible by the ARP interleaver (see chapter 3.7). Due to the small frame sizes, the problem of activity reduction is crucial.

6.2 Design of the turbo decoder

In this section, we first describe the architectural choices used to meet the requirements of the application. In the second part, we study the optimization of the implementation of the processor and pipelined processors. Finally, we will present the overall multi-processor turbo decoder architecture.

6.2.1 Architectural choices

The multi-processor decoder that has been designed uses the parallel architecture described in chapter 3. In particular, we saw in section 3.7 that the ARP interleaver enables a parallel architecture to be used, in a very flexible way. Since the interleaver is fixed, the only solution in order to maximize the activity is to use schedule $SW\text{-}\Sigma^*$ or schedule $SW\text{-}\Sigma^0$. In order to maximize the throughput we use the technique of pipeline multiplexing developed in chapter 5. To achieve a throughput up to 75 Mbit/s, we consider a turbo decoder with up to $P = 4$ pipelined processors:

- input quantization w_d: scalable ranging from 1 to 8 bits. The optimal quantization will be studied in section 6.4.
- window size L: a constant maximal size $L_{max} = 32$ is chosen since it represents a good trade-off between complexity and performance. For trellis sections lower than 32 symbols, the window size is reduced accordingly.
- initialization of recursions: usage of the Next Iteration Initialization (NII) technique (see chapter 1.5), which represents the best complexity versus performance trade-off.
- activity of the schedules: no overlapping of consecutive half-iterations is used. The activities of the two schedules $SW\text{-}\Sigma^*$ and $SW\text{-}\Sigma^0$ are compared in Figure 6-1 for 1, 2 and 4 physical processors with $L_{max} = 32$ and $l = 8$. The inflection observed on the curves comes from the fact that the size of the trellis section becomes lower than L_{max}. The comparison shows that the $SW\text{-}\Sigma^*$ architecture is the most efficient whatever the number of processors, which is consequently selected for our implementation. However, we saw in chapter 4.1 that schedule $SW\text{-}\Sigma^0$ offers the highest throughput.
- parallel accesses: since the period of the ARP interleaver is 4, up to 4 concurrent accesses to the memory banks can be performed at each cycle. The memory mapping described in chapter 3.7 is used: the data with the same congruence modulo 4 are mapped into the same memory bank. With the pipelined schedule $SW\text{-}\Sigma^*$ described in chapter 5.3, each physical processor performs a maximum of 1 read access and 1 write access per cycle. The architecture therefore includes one datapath for the read accesses and one datapath for the write accesses. The implementation of the datapaths for ARP interleavers was described in chapter 3.2 and 3.7.

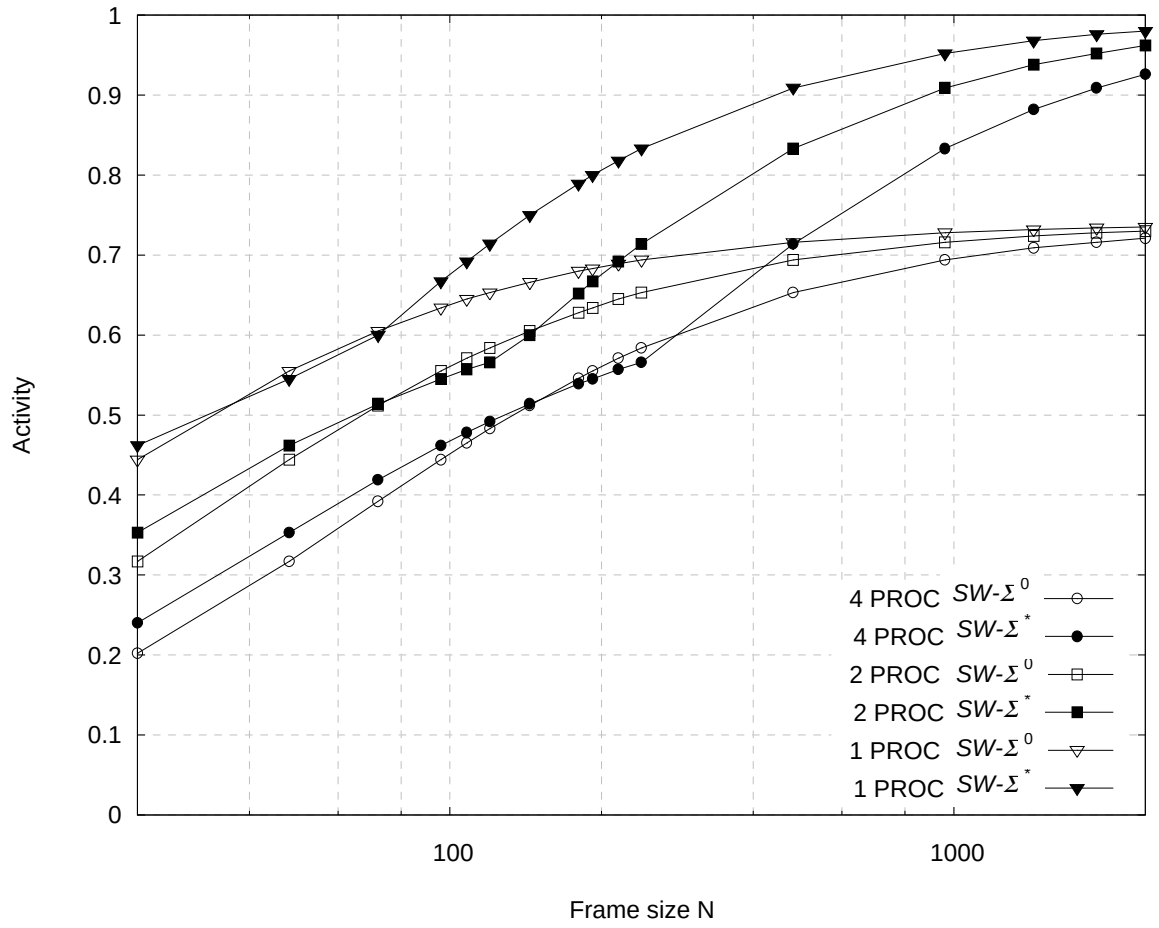


Figure 6-1: Activity comparison for schedules $SW\text{-}\Sigma^*$ and $SW\text{-}\Sigma^0$ for 1, 2 and 4 physical processors with $L_{max} = 32$ and $l = 8$

In the next section, we present an optimization of the processor for schedule $SW\text{-}\Sigma^*$.

6.2.2 Optimized processor architecture for schedule $SW\text{-}\Sigma^*$

The architecture for implementing schedule $SW\text{-}\Sigma^*$ presented in chapter 4.1 (see also Appendix A.3) uses several multiplexers that have a non-negligible impact on the complexity of the processors. In fact, multiplexers are required to multiplex the set of state metrics and the set of branch metrics. This section presents an architecture optimization technique that aims to suppress these multiplexers. In the first step, we consider a non-pipelined processor. The optimization of the pipelined processor will be addressed in the next section.

Instead of assigning one recursion unit (RU) to a forward or a backward recursion of one processor as in chapter 4.1, the RUs are shared between the two processors, as shown in Figure 6-2. The forward recursion unit FRU_1 is used to compute the forward state metrics during the first half of the windows, for each processor alternately. The corresponding state metrics are stored in the FSM MEM. Similarly, the second forward recursion unit FRU_2 computes the forward state metrics during the second half of the windows. Similarly, BRU_1 and BRU_2 are assigned to the first and second half of the windows, respectively. The recursions FRU_1 and BRU_1 use the inputs of the processor directly, while FRU_2 and BRU_2 read the inputs stored in the Extrinsic and Observation Sample memories EOS B and EOS F,

respectively. With these allocations, the soft output computation unit SOU F produces the soft outputs in the forward direction. It processes data of the first and the second processors, alternately. In the same way, SOU B computes the soft outputs in the backward direction. Using these assignations, no multiplexers are required inside the processor, they are rejected at the inputs and outputs of the processors, where there is no need for extra logic because multiplexing facilities already exist inside the datapaths for accessing the memory banks.

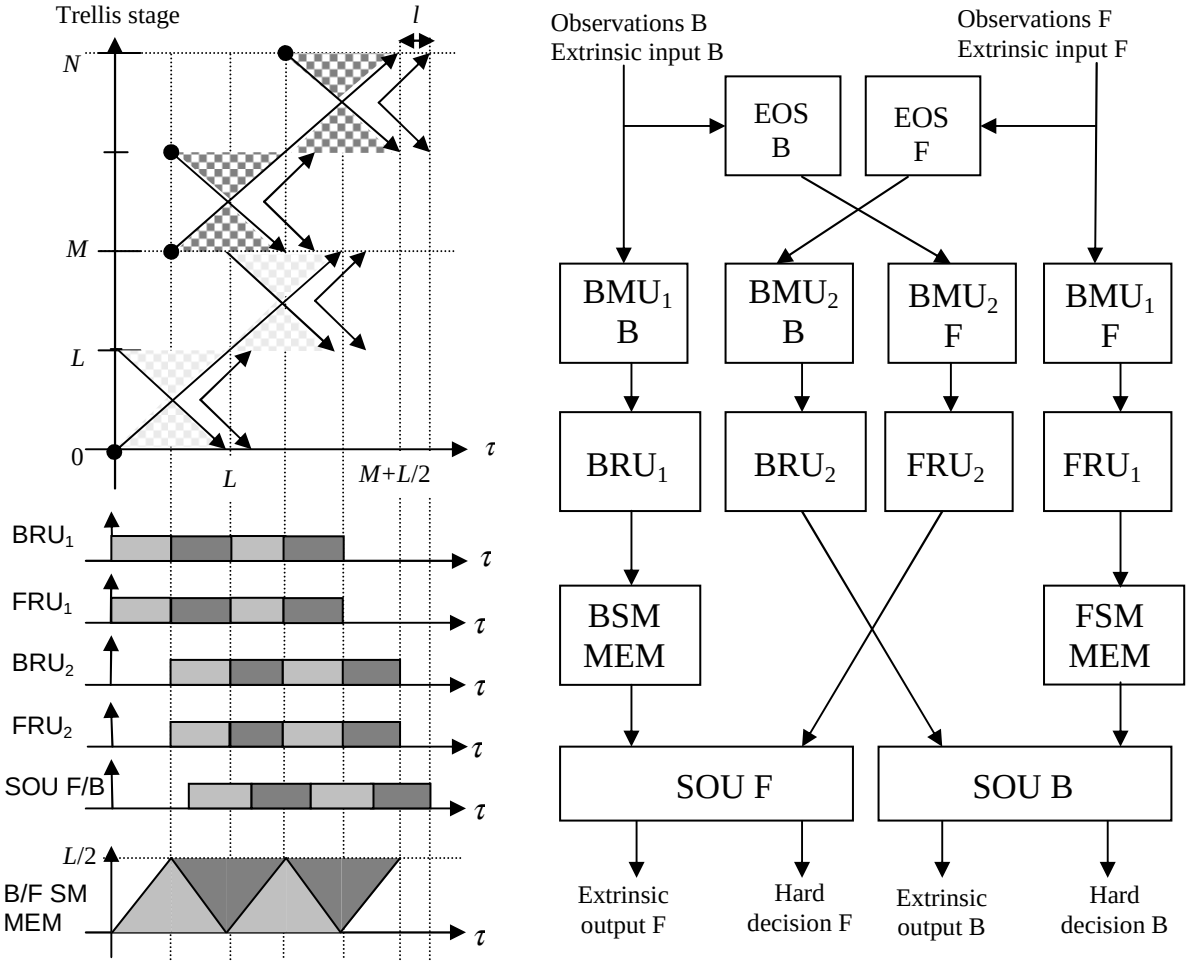


Figure 6-2: Optimized architecture and corresponding allocation for schedule *SW-Σ

In the next section, we apply a similar technique in order to optimize the pipelined processor for schedule *SW-Σ**.

6.2.3 Optimized pipelined processor architecture for schedule *SW-Σ**

In chapter 5.3, we described a pipelined processor for schedule *SW-Σ** (see also Appendix A.4), which includes a high number of multiplexers for multiplexing data between the two virtual processors (each virtual processor corresponds to one trellis section). It comprises a pipelined FRU (or BRU) processing alternately the first half and the second half of the window.

In order to reduce the number of multiplexers, a similar technique to that described in the previous section can be used. Merging FRU_1 and BRU_1 on the one hand and FRU_2 and BRU_2

on the other, leads to two pipelined recursion units (PRU), one processing the first half of the window, the other the second half. These pipelined units are used alternately for the first and second virtual processors. However, a single PRU works alternately in both the forward and backward directions. Thus, multiplexers are required inside the feedback loop in order to switch between the two directions (remember that forward and backward recursions have different ACS connectivity). This solution is not satisfactory because it increases the complexity and reduces the clock frequency.

In order to cope with the multiplexers inside the feedback loop, we have introduced a generic recursion unit (GRU) obtained by moving the multiplexers outside the feedback loop into the branch metric computation units and into the soft output computation unit. This solution, not described in any more detail in this manuscript, considerably reduces the number of multiplexers and allows a higher clock frequency. It was used to design the complete turbo decoder presented in the next section.

6.2.4 Design of the complete turbo decoder

The complete multi-processor turbo decoder is described in this section. It is composed of three modules as depicted in Figure 6-3:

- input module: receives the input frames and transmits them to the decoder module. It comprises a double input buffer, in order to receive the next frame while decoding the current one. The input buffer is divided into as many memory banks as the number of physical processors.
- decoder module: performs I_{max} iterations on the frame stored in the input module and writes the decoded codeword into the output module. This module contains up to $P = 4$ pipelined processors and an extrinsic memory decomposed into as many memory banks as the number of physical processors (not represented in the figure). A single Finite State Machine (FSM) (not represented) controls the pipelined processors.
- output module: stores the hard decisions produced by the decoder module and sends them to the output of the decoder.

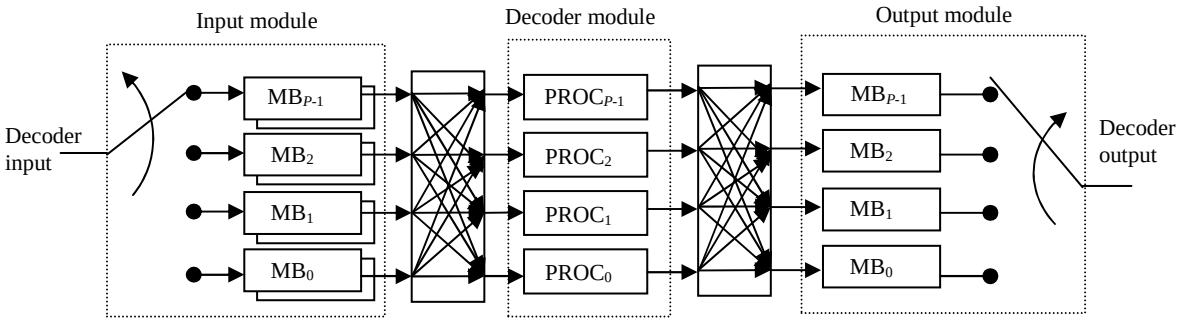


Figure 6-3: Complete turbo decoder description for $P = 4$

6.3 Synthesis results

The multi-processor turbo decoder has been designed for various FPGA circuits including Altera Stratix II and Cyclone II, Xilinx Virtex II, Spartan 3 and Virtex 4. In this section, we present only the results obtained for an Altera Stratix II FPGA. We have used device EP2S30 with the fastest speed grade C3. Table 6-2 presents the synthesis results (maximal working frequency and complexity) obtained for various numbers of input quantization bits and numbers of processors. The synthesis results have been obtained after Place and Route with Quartus II 4.1.

The synthesis results shows that increasing the number of quantization bits or the number of processors obviously leads to an increase in the complexity, but also to a reduction in the maximal clock frequency. This reduction in clock frequency is related to the greater number of resources used, making the place and route operation more difficult. These results will be used in the next section, in order to analyze the performance of the architectures.

Number of pipelined processors P	Quantization w_d	Maximal clock frequency f_{clk} (MHz)	Complexity (logic and memory blocks)		
			ALUTs	M4K	M512
1	3	230	5359	40	3
	4	222	5782	50	2
	5	216	6144	59	8
	6	218	6528	74	3
2	3	210	9002	56	3
	4	210	9789	60	3
	5	203	10617	75	4
	6	197	11499	81	2
4	3	193	15849	79	5
	4	190	17750	82	5
	5	185	19555	85	35
	6	184	21293	116	3

Table 6-2: Synthesis results of the multi-processor turbo decoder for a Stratix II circuit

6.4 Discussion of performance

In this section, we first recall the basic equations used to characterize the architectures: the activity, the throughput, and the decoding delay:

$$\alpha_{SW-\Sigma^*} = \frac{N/(2P)}{N/(2P) + L/2 + l} \quad \textbf{(6-1)}$$

$$D_b = P \cdot \frac{f_{clk} \cdot \alpha}{I_{\max}} \quad \textbf{(6-2)}$$

$$d_{TD} = \frac{2I_{max}}{f_{clk}}(N/P + L + 2l) \quad \textbf{(6-3)}$$

Although the activity is a suitable measure to compare the efficiency of various architectures and schedules, it does not take into account the number of quantization bits of the variables in the decoder. Thus, in order to compare the influence of the quantization on the efficiency of the architecture, we will use the ratio throughput versus complexity.

In the next sections, we analyze the impact of the number of processors and quantization bits on the error decoding performance. This study does not take into account the complexities of the architectures, which is used in the last part in order to evaluate the impact of quantization on the architecture efficiency. Finally, the simulated performance curves guaranteeing a minimal throughput are given.

6.4.1 Discussion of the number of processors

Figure 6-4 presents the error decoding performance for the whole range of frame sizes with a QPSK modulation and code rate 1/2. It gives the value of E_b/N_0 required to reach a FER of 10^{-2} for various numbers of processors with $w_d = 6$ and $I_{max} = 8$. The performance is compared to the optimal performance obtained with $w_d = 8$ and using pre-processing steps of length $L' = 32$ (see chapter 1.5) to initialize the recursions. The performance curves show that with a single processor the NII technique is associated with a degradation of less than 0.1 dB compared to the optimal performance. This clearly demonstrates the efficiency of the NII technique. In addition, increasing P does not degrade the performance of large frame sizes. For smaller frame sizes, a degradation remaining below 0.3 dB is associated with the increase in P, which is mainly explained by the reduction in L.

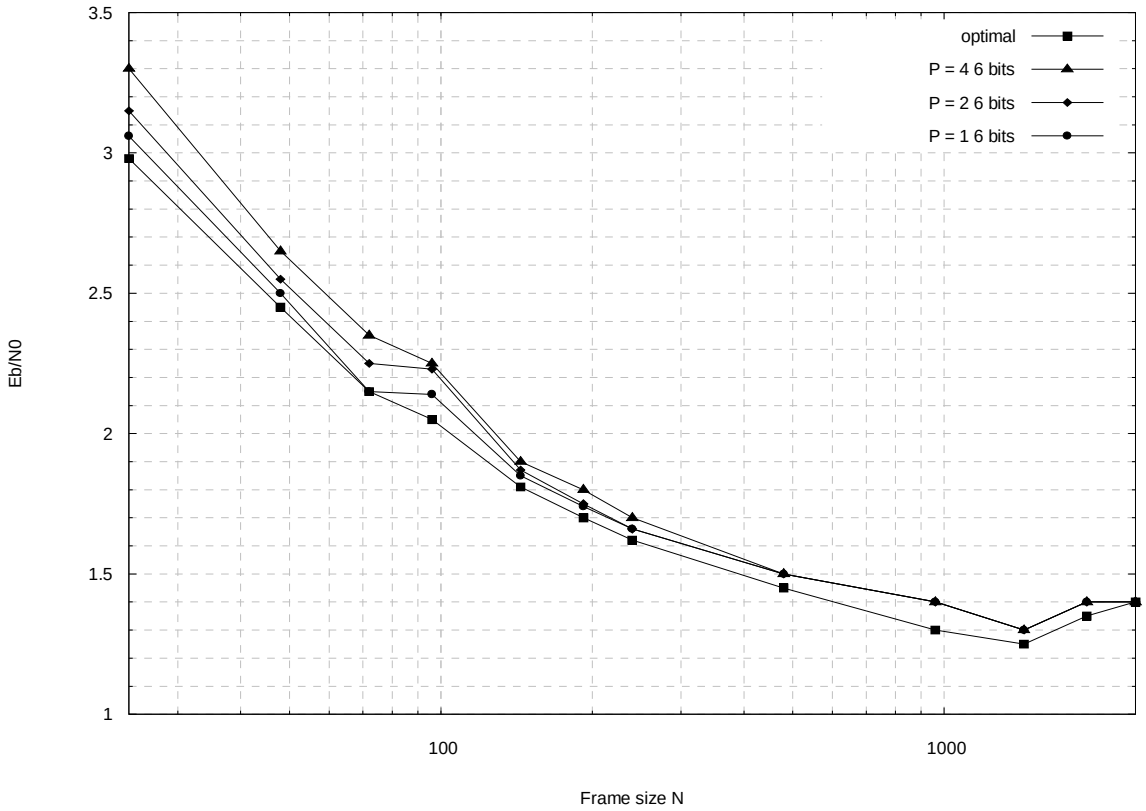


Figure 6-4: Performance for an FER of 10^{-2} as a function of the number of processors for w_d = 6 quantization bits at I_{max} = 8 iterations

6.4.2 Discussion of the quantization width

Figure 6-5 presents the error decoding performance of the whole range of frames for $P = 4$ pipelined processors. It compares the value of E_b/N_0 required to reach an FER of 10^{-2} at $I_{max} = 8$ for various w_d ranging from 3 to 6 bits (modulation QPSK, code rate 1/2). This study shows that the performance for $w_d = 5$ is equivalent to the performance with $w_d = 6$. Then, decreasing the quantization leads to a constant degradation over the range of frame sizes. The degradation equals around 0.1 dB for $w_d = 4$ and around 0.4 dB for $w_d = 3$. The case $w_d = 3$ is thus not very attractive. In order to compare the most efficient quantization schemes, we need to take into account the complexities of the respective architectures. This is addressed in the next section.

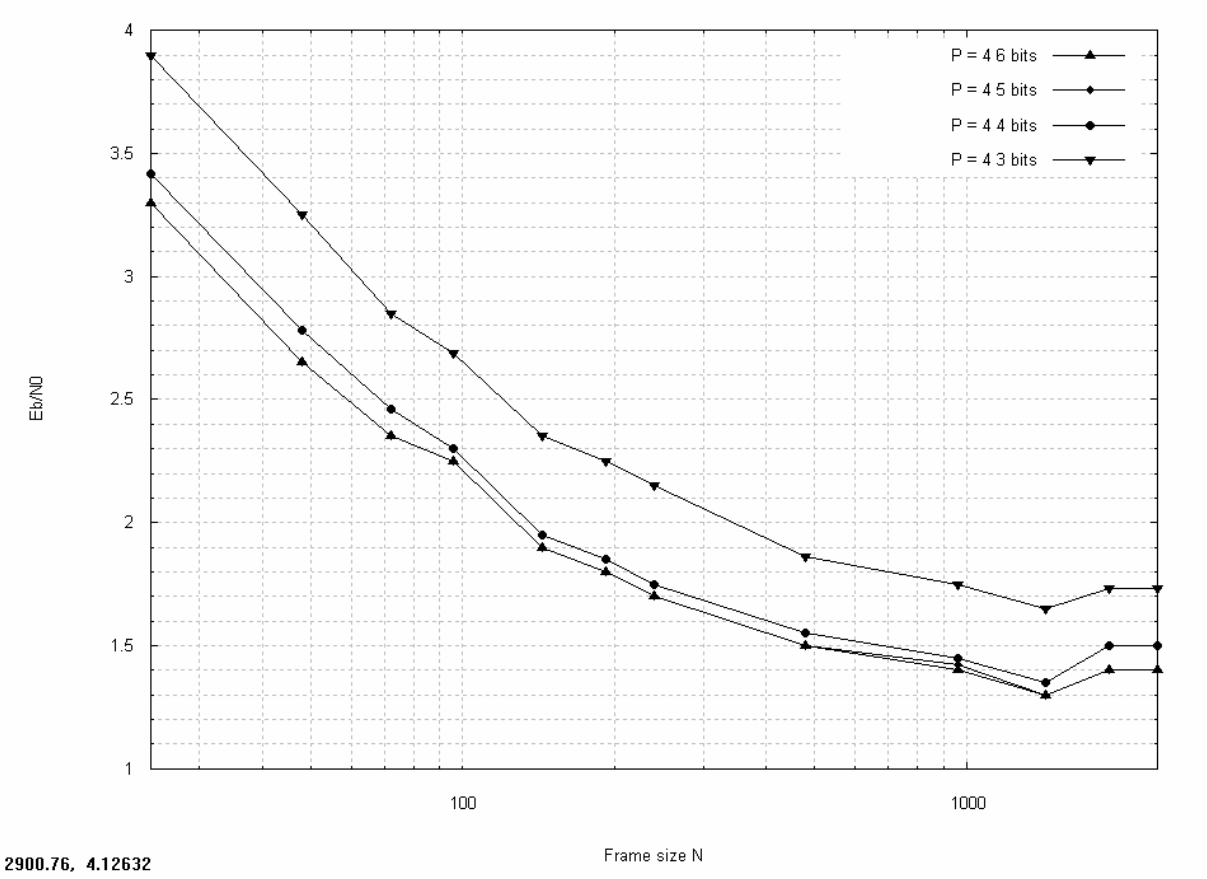


Figure 6-5: Performance for an FER of 10^{-2} as a function of the number of quantization bits for $P = 4$ processors at $I_{max} = 8$ iterations

6.4.3 Impact of the quantization on the ratio performance / complexity

In this sub-section, we take into account the complexities and the maximal clock frequencies given in section 6.3 to evaluate the best quantization trade-off in terms of performance versus complexity. To this end, we first give the required E_b/N_0 required to reach a FER of 10^{-2} as a function of the number iterations for w_d ranging from 3 to 6. Figure 6-6 and Figure 6-7 show the curves obtained for N = 120, 24 and 1440. Three important observations can be made:

- for a given I_{max}, the degradation associated with a reduction in w_d is constant over the whole range of I_{max}. In particular, asymptotically (with a high number of iterations), the performance for low w_d remains lower than the performance for high w_d.
- for a given target E_b/N_0, the difference between the number of iterations for two different w_d is not constant over the range of E_b/N_0. At a high E_b/N_0, the difference in the number of iterations is low, while at low E_b/N_0 the difference is greater.
- degradation with lower w_d depends on the size of the frame: it is lower with large N, and higher with shorter N.

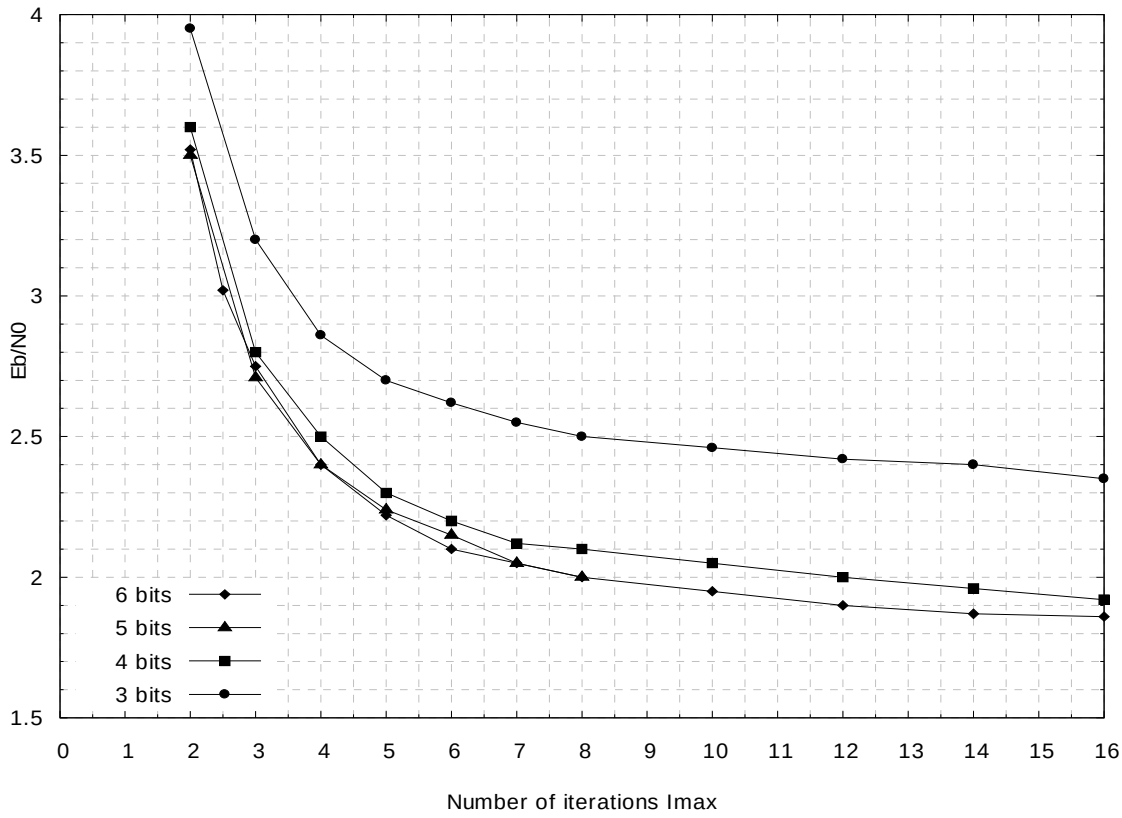


Figure 6-6: E_b/N_0 at an FER of 10^{-2} as a function of the number of iterations for w_d = 3, 4, 5, 6 for N = 120

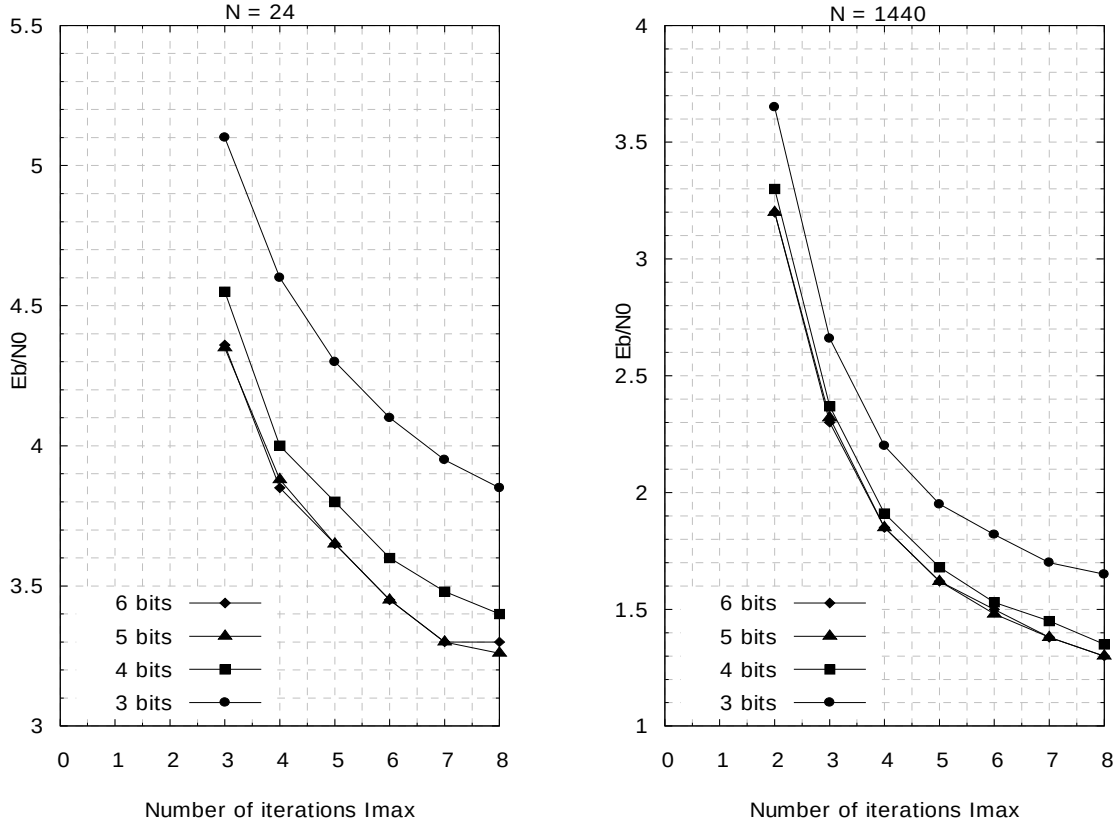


Figure 6-7: E_b/N_0 at an FER of 10^{-2} as a function of the number of iterations for w_d = 3, 4, 5, 6 for N = 24 and N = 1440

The curves given in the two preceding figures are used to perform the comparison of architecture efficiency as a function of w_d. The architecture efficiency is defined as the ratio of the throughput of the architecture (related to the clock frequency given in Table 6-2) given by (6-2) divided by the complexity expressed in number of ALUTs and given in Table 6-2. The complexity does not take into account the number of memory blocks.

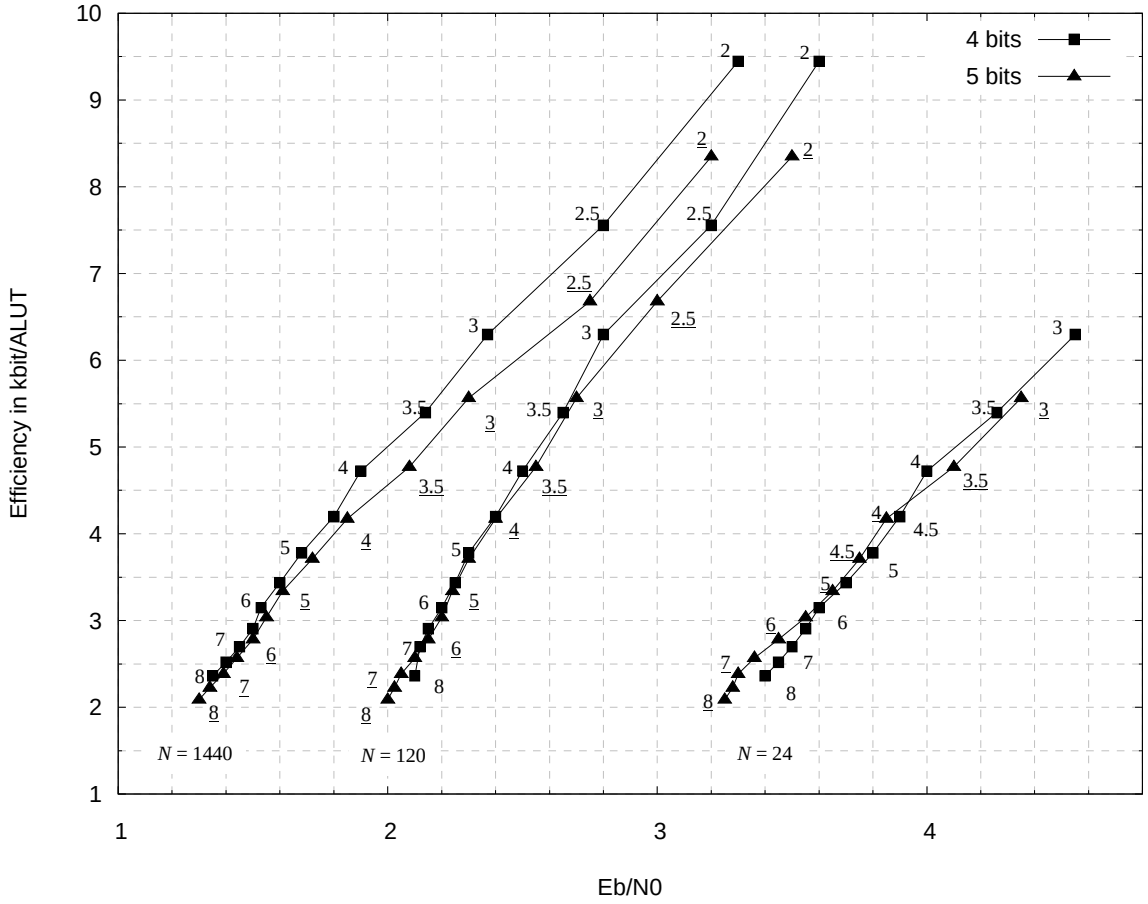


Figure 6-8: Efficiency of the architecture as a function of the E_b/N_0 required for an FER of 10^{-2}

Figure 6-8 gives the efficiency of the architecture as a function of E_b/N_0 at an FER of 10^{-2} for $P = 4$ pipelined processors. Only the curves for $w_d = 4$ and $w_d = 5$ are represented, since the architectures for $w_d = 3$ and $w_d = 6$ are always less efficient. The number of iterations are also given on these curves (underlined for $w_d = 5$). This figure shows that in order to reach a target E_b/N_0 with a given frame size, an architecture with either $w_d = 4$ or $w_d = 5$ can be used However, it does not necessarily have the same efficiency, due to the difference required in the number of iterations. For example, with a frame size of $N = 1440$, $E_b/N_0 = 2.3$ is reached either with 2 iterations and $w_d = 5$ or with 2.5 iterations and $w_d = 4$. The efficiencies are 6.6 and 7.5 kbit/ALUT, respectively. From this figure, the following general trends can be obtained:

- for a fixed frame size, $w_d = 4$ is more efficient for low I_{max} or high E_b/N_0, while $w_d = 5$ is more efficient for high I_{max} or low E_b/N_0.

- the region where the two architectures are equivalent depends on the frame size: around 4 iterations for short to medium N, around 6 iterations for large N.

Therefore, the choice between $w_d = 4$ or $w_d = 5$ depends on the number of iterations performed by the decoder, which is related to the target throughput of the decoder. In addition, because of the frame size-dependent activity, I_{max} increases with N. Thus, to reach a target throughput of 75 Mbit/s, only 2 iterations are allowed for $N = 24$. This corresponds to the regions where the architecture with $w_d = 4$ is more efficient. For $N = 120$ and $N = 1440$, around 4 and 9 iterations, respectively, are performed, thus the two architectures are equivalent in terms of efficiency. For a much higher throughput, the architecture with $w_d = 4$ would always be more efficient.

6.4.4 Performance at constant throughput

For $P = 4$ pipelined processors, the two architectures with $w_d = 4$ and $w_d = 5$ are almost equivalent, but the architecture with $w_d = 4$ uses less memory resources. This solution is thus selected in order to simulate the performance of the decoder for a minimal throughput of $D_b = 75$ Mbit/s. The performance curves are given in Figure 6-9.

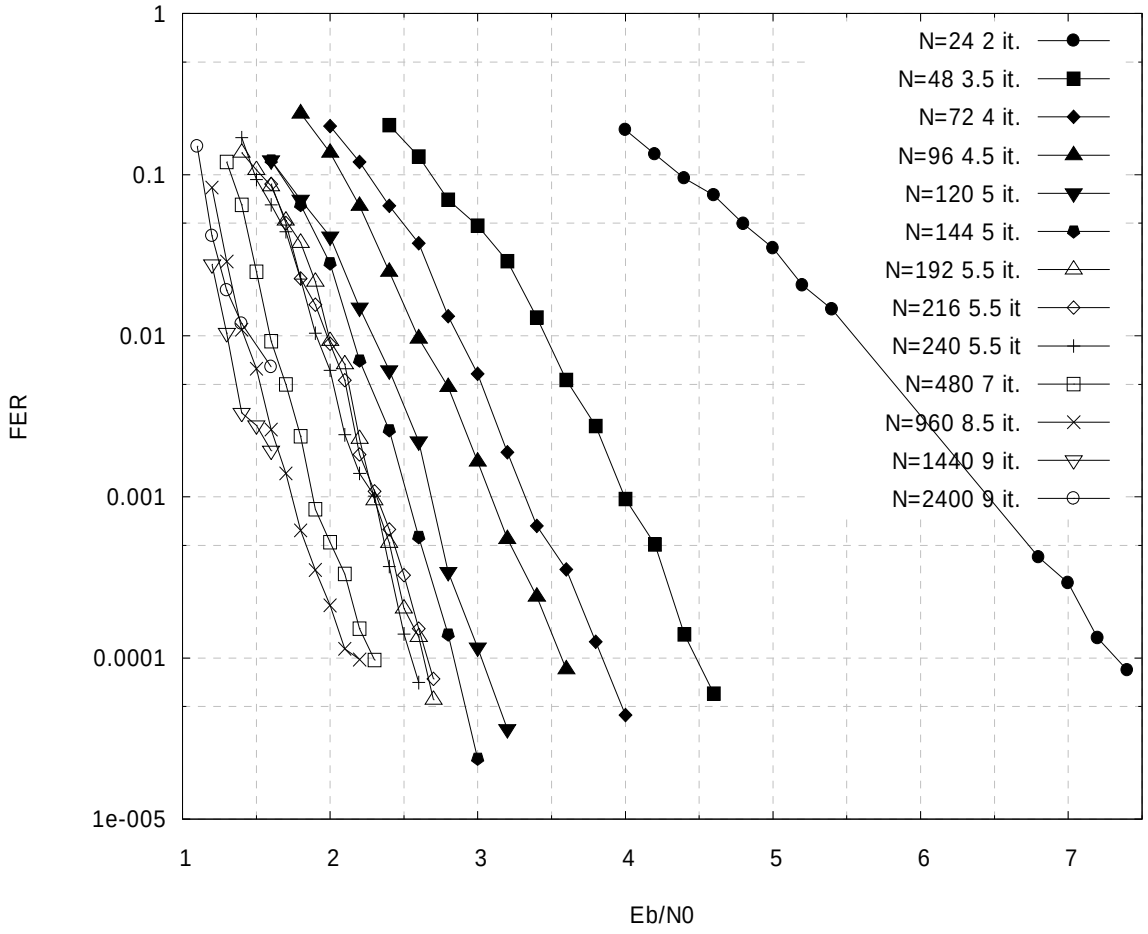


Figure 6-9: Performance of the decoder for a minimal throughput D_b = 75 Mbit/s with P = 4 and w_d = 4

6.5 Conclusion

In this chapter we have used some of the techniques described throughout this manuscript to design a turbo decoder implementation on an FPGA for the WiMAX standard. Since the turbo decoder is associated with a double-binary turbo code with an ARP interleaver, our architecture is composed of up to 4 pipelined processors working in parallel and performing schedule $SW\text{-}\Sigma^*$. This architecture covers a large range of frame sizes from 24 to 2400 double-binary symbols.

The turbo decoder has been designed for various FPGA circuits including low-cost FPGAs (Cyclone II, Spartan 3) and high performance FPGAs (Stratix II, Virtex 4). The synthesis results given for a Stratix II FPGA demonstrate that a high clock frequency and a relative low complexity can be reached for 4 processors and $w_d = 4$.

The simulated error decoding performance has shown that, for a given number of iterations, increasing the number of processors leads to reasonable performance degradation for small frame sizes only (below 120 couples) bounded by 0.3 dB. At constant throughput, the major performance degradation is caused by the activity reduction for small frame sizes that obliges the number of decoding iterations to be reduced. Our study of the efficiency of the architecture has demonstrated that quantization on 4 or 5 bits of the decoder inputs is the most efficient. The optimal choice between 4 and 5 bits depends on the target throughput of the decoder. Finally, we have simulated the error decoding performance of the 4-processor decoder matched to a constant decoded throughput of 75 Mbit/s. The curves have been given for a QPSK modulation and a code rate 1/2. For a frame size of 240 bits, the decoder reaches an FER of 10^{-2} at $E_b/N_0 = 2.25$ dB at a throughput of 75 Mbit/s.

In conclusion, the turbo decoder implementation obtained in this chapter validates the effectiveness of the techniques proposed in this thesis to maximize the efficiency of the architecture. The decoder reaches a throughput of several tenths of a Mbit/s with excellent complexity and error decoding performance. These results lead to high-speed, high-performance and low-cost turbo decoder implementations.

Conclusions and perspectives

Conclusion and perspectives

Since their discovery in the beginning of the nineties, turbo codes and the associated iterative decoding principle have spread widely over numerous telecommunications applications. Nowadays, iterative-decodable codes like turbo codes are the best performing channel codes, as they almost reach the Shannon limit.

Turbo codes encompass a large family of error correcting codes designed as a concatenation of several constituent codes separated by interleavers. This concatenation structure and the concept of extrinsic information enable these codes to be decoded iteratively using Soft-Input Soft-Output decoders. Due to their excellent performance they have been introduced widely in recent telecommunications standards for satellite and wireless applications. Low throughput (below 10 Mbit/s) turbo decoders have already been implemented, either in software or hardware, by several research teams and companies, and are available commercially. With the emergence of broadband access, and especially the recent introduction of turbo codes into the WiMAX standard, there is an increasing demand for high-throughput and low-cost decoders. This thesis lies within this general scope of throughput augmentation. In particular, the cost constraint imposes the design of efficient architectures exploiting the possibilities of the circuit to its maximum. We have concentrated our work on parallel concatenations of RSC codes, especially double-binary codes, but most of the results obtained are generally applicable to other concatenations of convolutional codes.

After outlining the principles of turbo codes and their associated iterative decoding principle, we have recalled the optimal Maximum A Posteriori soft output decoding algorithm for m-binary RSC codes, yielding its simplified expression in the logarithmic domain: the Max-Log-MAP algorithm. Sliding window computation schedules enabling memory and latency saving have been reviewed. We have then stripped unnecessary computations from the algorithm using the Next Iteration Initialization technique for initializing the forward and backward recursions. We have then proposed a simple general model for expressing the throughput and the efficiency of an architecture. The application of this model to the above-mentioned stripped algorithm has highlighted three research fields that have enabled us to reach three significant milestones in our quest towards efficient high-throughput turbo decoder architectures.

Our first contribution concerns parallel decoding architectures and the resulting parallel conflicts resulting from concurrent accesses to the memory banks. In order to avoid the duplication of memories in serial architectures, parallel architectures have been considered for improving the throughput of turbo decoders. We have thus classified the known techniques to remedy this problem as "execution", "compilation" and "design" stage solutions, depending on the stage at which they tackle the problem. Our contribution operates at the latter stage during the design of the code and particularly the interleaver. We proposed a joint interleaver-architecture design methodology based on a new family of turbo codes, Multiple Slice Turbo

Codes. MSTCs are designed as a concatenation of several Circular RSC codes (slices) associated with a hierarchical interleaver design. We validated by simulation that the introduction of constraints in the interleaver design introduces no performance degradation. We have also presented another joint interleaver-architecture construction based on the ARP interleaver that has greater flexibility and has the crucial advantage of having already been introduced in standards such as DVB-RCS and IEEE 802.16 (WiMAX).

Due to the huge activity reduction associated with parallel architectures our second contribution involves the investigation of solutions to overcome this. In the first step, we have shown that the activity of the architecture is strongly related to the schedule of the Max-Log-MAP algorithm. In line with this statement we have improved existing solutions by introducing a schedule adapted to parallel decoding in which computation units are shared between two processors. But, due to the constant overhead in the schedules, additional improvements have been taken into consideration, especially for short frame size and / or high number of parallel processors. In the second step, we have proposed a hybrid architecture combining the advantages of the parallel and the serial architectures. This solution induces modifications at system level since the overall latency is not linear with the size of the frame. In the third step, we have proposed a solution enabling a significant increase in activity in the context of iterative decoding. It is based on overlapping in the processing of consecutive half iterations, in order to reduce the time when the processing units are idled. This modified algorithm is sub-optimal since it leads to the usage of a great proportion of extrinsic data conflicts that are not up-to-date, thus slowing down the convergence of the iterative process. This latter configuration is related to what we call consistency conflicts. The proportion of consistency conflicts is related to the association of the interleaver and the parallel schedule. After an analytical evaluation of this proportion of consistency conflicts, a low complexity architecture implementing this sub-optimal algorithm has been described. Finally, we have constrained the interleaver in association with the parallel decoding schedule in order to be able to decode with maximal activity (using half-iteration processing overlapping) and no consistency conflicts, *i.e.* usage of extrinsic information that has already been produced. We have validated by simulation the efficiency of this approach using MSTCs associated with a three-level hierarchical interleaver. These results have not yet been published.

At implementation level, an important breakthrough has been achieved by breaking the "ACS bottleneck" induced by the recursive computation of state metrics. This solution involves multiplexing several state metric recursions belonging to different trellis sections on a single pipelined recursion unit. The core element of our contribution to this technique concerns the way we exhibit the trellis sections: we have proposed to associate it with the trellis sections obtained with the parallel architecture. In particular, we have described the insertion of a pipeline register into a recursion unit implementing the Max-Log-MAP algorithm for double-binary RSC codes and enabling a doubling of the clock frequency. The proposed scheme has been successfully applied to the implementation of several schedules described in this thesis.

Finally, we have described an FPGA implementation of a multi-processor decoder for the WiMAX standard. This implementation includes and validates several improvements provided throughout this thesis: parallel architectures, improved schedule for high activity, pipeline multiplexing technique, etc. This implementation integrates up to four pipelined processors and achieves a throughput up to 100 Mbit/s at 6 iterations.

Above all, an important contribution of this thesis is to show the interest in reversing the sequence order of the conventional approach to code design. Until now, the "Holy Grail" has been to construct the "best" code, but little consideration has been given to its hardware implementation. We have shown that designing the hardware architecture prior to, or at least jointly with, the choice of the code is a key that leads to efficient high-speed turbo decoders.

Perspectives:

Short term perspectives of our work in the context of turbo decoder implementations are related to the augmentation of the activity for parallel architectures. Important work still needs to be done at the system level in order to optimize the hybrid structure. In addition, we believe that the sub-optimal algorithm handling consistency conflicts during the execution of an overlapping schedule can be improved. Finally, the joint interleaver-architecture methodology proposed for using only up-to-date extrinsic information opens up new directions for interleaver optimization. The application to conventional turbo code design (with no slices) is of particular interest.

More **general perspectives** are related to the application of the above-mentioned reversed philosophy to other error correcting codes such as LDPC codes. More generally, the application to other iterative-decodable schemes is of great interest.

Appendix A

Max-Log-MAP decoding schedules

This appendix summarizes the decoding schedules and the corresponding architectures used to perform the Max-Log-MAP algorithm.

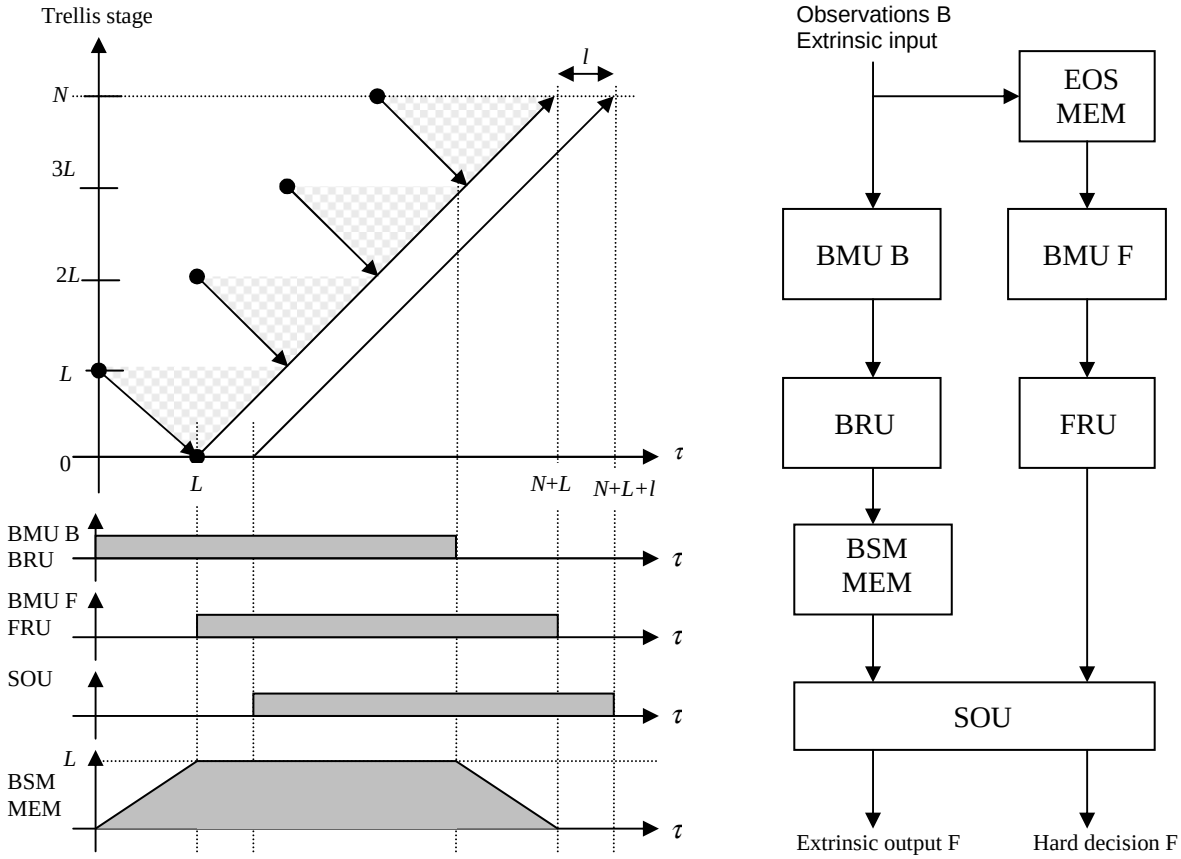
Content:

Appendix A. Max-Log-MAP decoding schedules

A.1 Schedule *SW-Σ⁻*

Activity of schedule *SW-Σ⁻* for *P* processors:

$$\alpha_{SW-\Sigma^-} = \frac{N/P}{N/P + L + l}. \qquad \textbf{(A-4)}$$



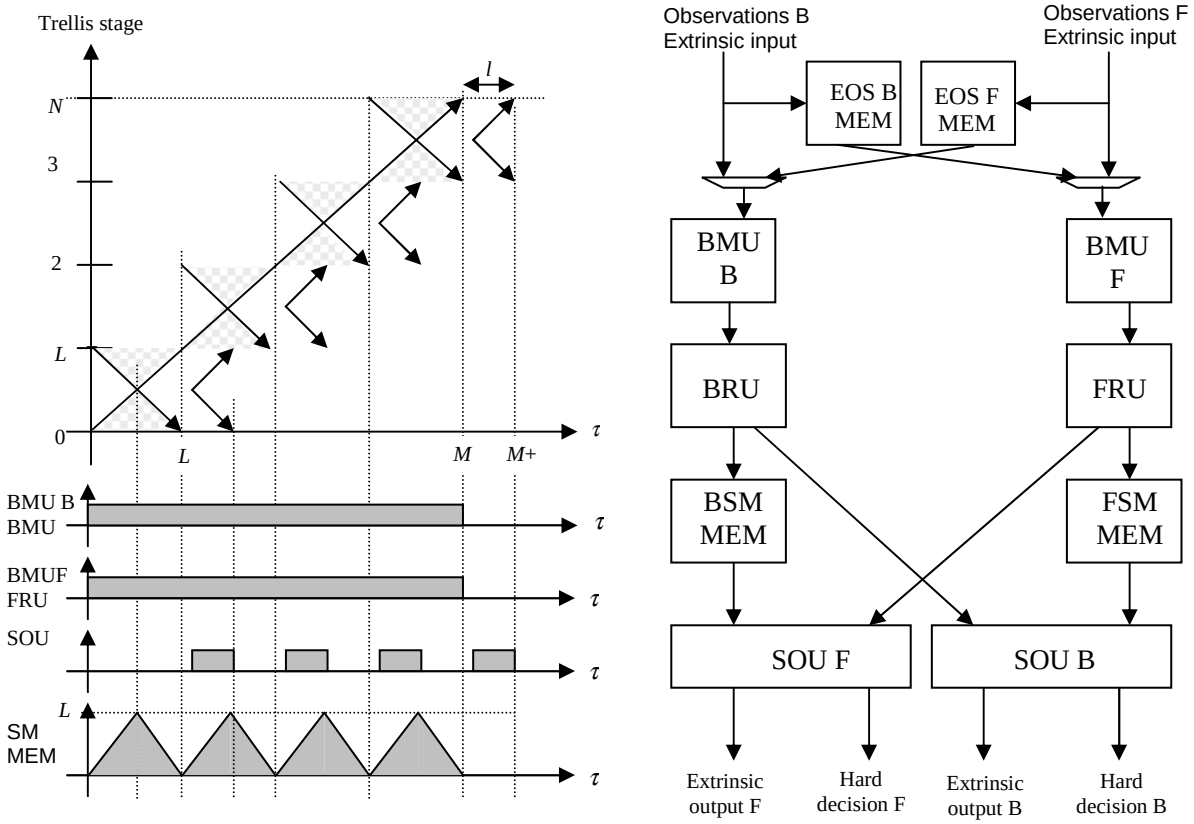
EOS MEM : Extrinsic and Observation Samples memory.
BSM MEM : Backward State Metric memory.
BMU F : Forward Branch Metric Unit.
BMU B : Backward Branch Metric Unit.
BRU : Backward Recursion Unit.
FRU : Forward Recursion Unit.
SOU : Soft Output Unit.

A.2 Schedule *SW-*Σ^0

Activity of schedule *SW-*Σ^0 for *P* processors:

$$\alpha_{SW-\Sigma^0} = \beta_{SW-\Sigma^0} \cdot \frac{N/P}{N/P + l}, \qquad \textbf{(A-5)}$$

where $\beta_{SW-\Sigma^0}$ = 0.75.

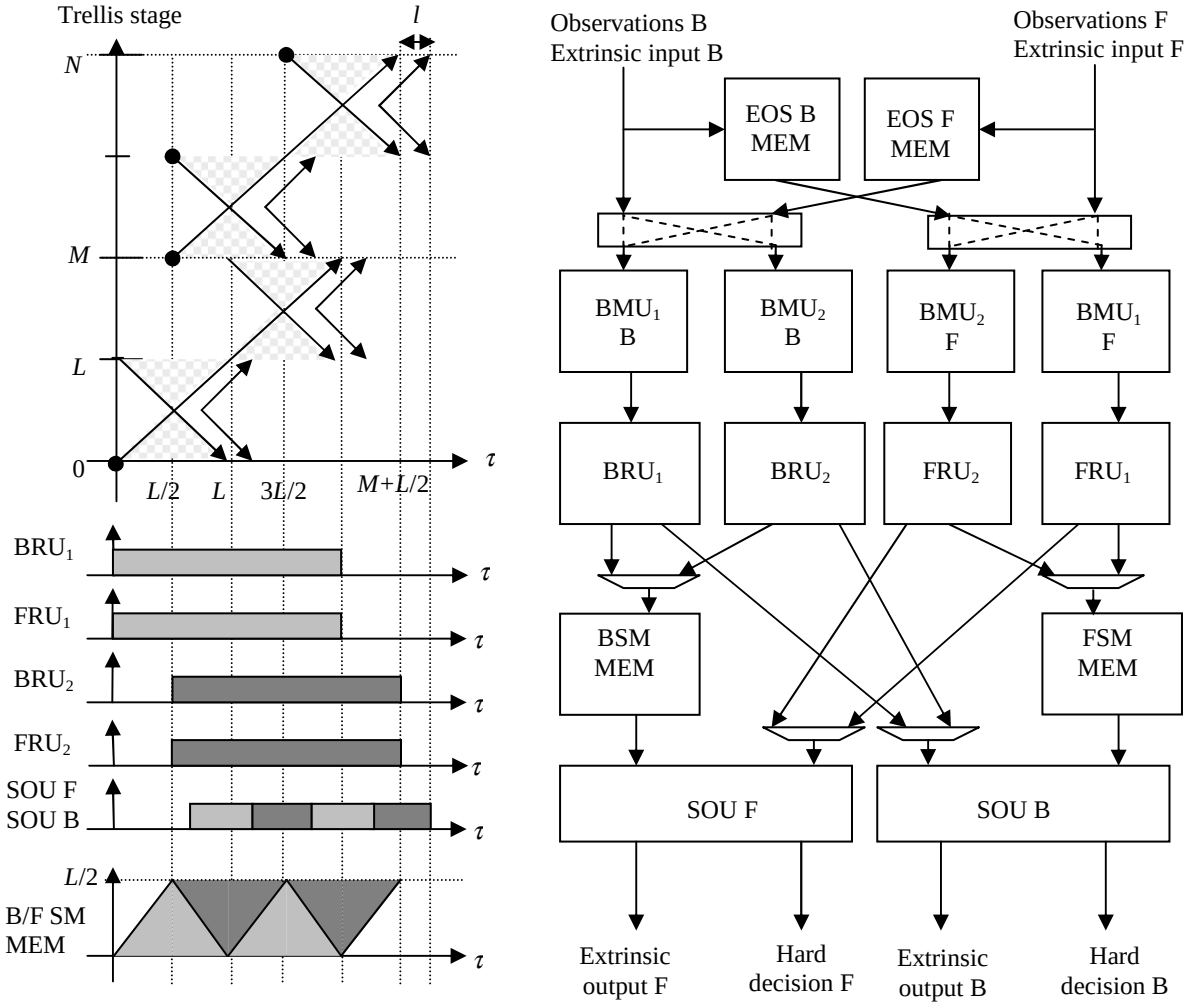


EOS MEM B : Backward Extrinsic and Observation Samples memory.
EOS MEM F : Forward Extrinsic and Observation Samples memory.
BSM MEM : Backward State Metric memory.
FSM MEM : Forward State Metric memory.
BMU F : Forward Branch Metric Unit.
BMU B : Backward Branch Metric Unit.
BRU : Backward Recursion Unit.
FRU : Forward Recursion Unit.
SOU F : Forward Soft Output Unit.
SOU B : Backward Soft Output Unit.

A.3 Schedule *SW-Σ**

Activity of schedule *SW-Σ** with $P \geq 2$ processors:

$$\alpha_{SW-\Sigma^*} = \frac{N/P}{N/P + L/2 + l}. \qquad \textbf{(A-6)}$$



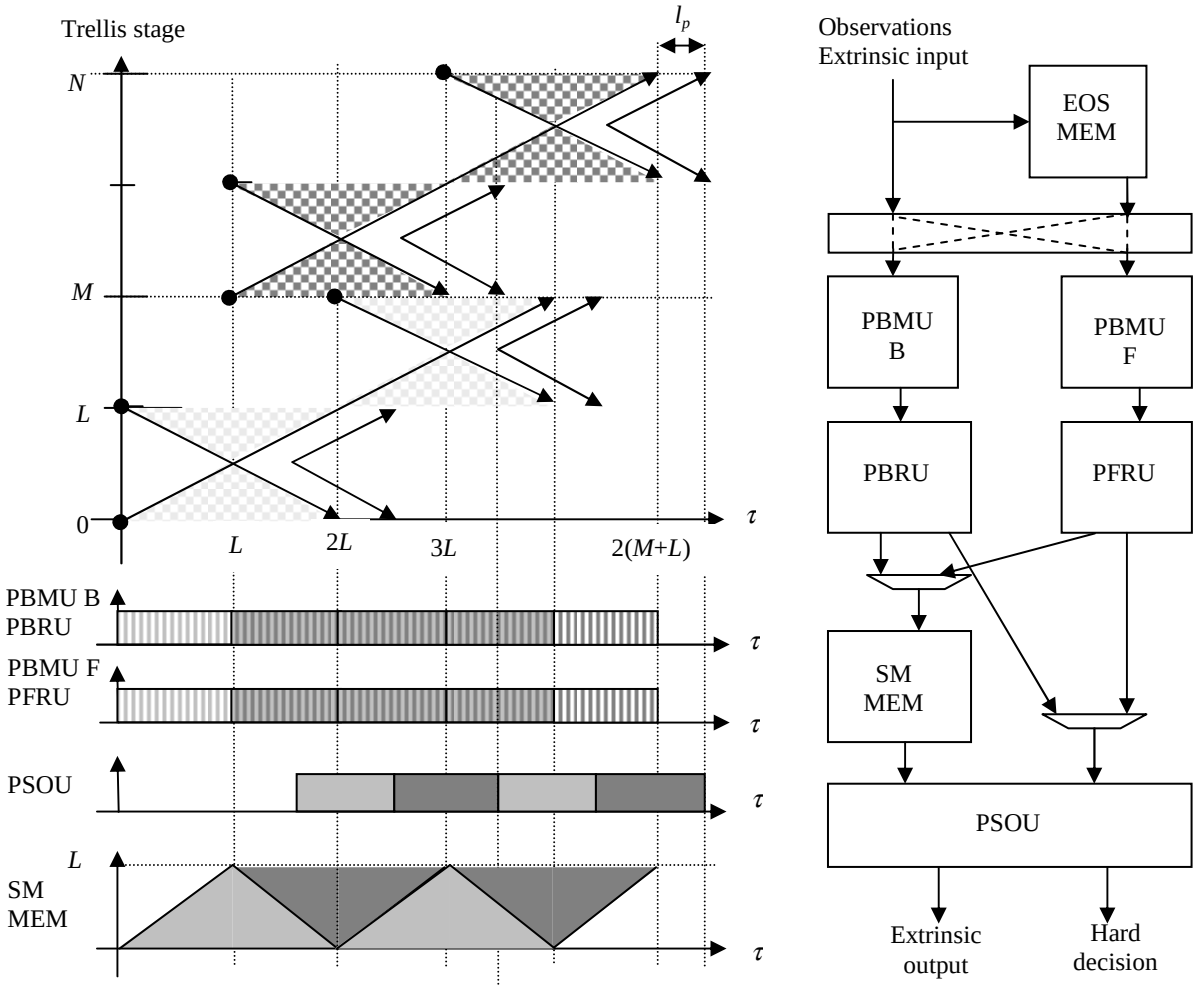
EOS MEM B	: Backward Extrinsic and Observation Samples memory.
EOS MEM F	: Forward Extrinsic and Observation Samples memory.
BSM MEM	: Backward State Metric memory.
FSM MEM	: Forward State Metric memory.
BMU F	: Forward Branch Metric Unit.
BMU B	: Backward Branch Metric Unit.
BRU	: Backward Recursion Unit.
FRU	: Forward Recursion Unit.
SOU F	: Forward Soft Output Unit.
SOU B	: Backward Soft Output Unit.

A.4 Schedule *SW-Σ** for a pipelined processor

Activity of schedule *SW-Σ** with $P \geq 1$ pipelined processor:

$$\alpha_{SW-\Sigma^*} = \frac{N/(2P)}{N/(2P) + L/2 + l}. \qquad \textbf{(A-7)}$$

where $l = l_P / 2$.



EOS MEM : Extrinsic and Observation Samples memory.
SM MEM : State Metric memory.
PBMU F : Pipelined Forward Branch Metric Unit.
PBMU B : Pipelined Backward Branch Metric Unit.
PBRU : Pipelined Backward Recursion Unit.
PFRU : Pipelined Forward Recursion Unit.
PSOU : Pipelined Soft Output Unit.

Appendix B

Maximal Spread of an Interleaver

This appendix gives a geometrical proof of the maximal spread of an interleaver. The maximal spread for a Multiple Slice Turbo Code (MSTCs) is also computed. The material presented in this chapter has been published in [Boutillon et al.-05]. For the generalization to the multi-dimensional case we refer the interested reader to [Boutillon et al.-05].

Content:

Appendix B. Maximal spread of an Interleaver

After a brief review of the spread definition in the context of turbo-code design, the method used for the derivation of an upper bound of the maximum spread for two dimensional conventional single slice turbo codes (SSTCs) is explained. Then, the generalization to MSTCs is presented.

B.1 Definition of spread

The concept of spread to design the so-called "S-random" interleavers was introduced in [Dolinar *et al*.-95] for non-tail-biting codes. The spread definition was modified in [Crozier *et al*.-00] and was also extended to the case of tail-biting codes (although this is not stated explicitly). In this paper, the spread definition used in [Crozier *et al*.-00] is explicitly stated for tail-biting codes and is extended to account for a variable number of dimensions. Let N denote the length of the information block of the code. In order to simplify the notation, the block length will not be mentioned when there is no ambiguity on its value. Let Π denote the permutation function that associates an index $\Pi(k)$, in the interleaved order, to an index k in the natural order.

Definition B-1: In the 2-dimensional case, the spread between two symbols k_1 and k_2 is defined as $S_2(k_1,k_2) = |k_1 - k_2|_N + |\Pi(k_1) - \Pi(k_2)|_N$, where $|a-b|_N$ is equal to $\min(|a-b|, N-|a-b|)$.

This definition can be easily generalized to the D-dimensional case by defining the D-dimensional vector $\Phi(k) = (\Pi_0(k), \Pi_1(k)..., \Pi_{D-1}(k))$ associated to the symbols k. The vector $\Phi(k)$ represents the indices of the symbol k in the D interleaved dimensions. For the non-interleaved order (dimension 0), $\Pi_0 = Id$, where Id stands for the Identity function. Let $\mathbf{Z}$ be the set of integers and let $E = \mathbf{Z}/N\mathbf{Z} = \{0,..., N-1\}$, be the ring of integer numbers modulo N. By definition, we have $k \in E$ and $\Phi(k) \in E^D$.

Definition B-2: In the D-dimensional case, the spread $S_D(k_1,k_2)$ between two symbols k_1 and k_2 is defined as an application from $E \times E$ into $\mathbb{R}^+$:

$$S_D(k_1,k_2) = \sum_{i=0}^{D-1} |\Pi_i(k_1) - \Pi_i(k_2)|_N \ . \qquad \textbf{(B-1)}$$

Definition B-3: The spread S_D of a D-dimensional interleaver $\Pi = (\Pi_0, \Pi_1..., \Pi_{D-1})$ is the minimum spread between all couples of different symbols:

$$S_D = \min\left\{S_D(k_1,k_2), k_1,k_2 \in E^2, k_1 \neq k_2\right\}. \quad \textbf{(B-2)}$$

Definition B-4: The maximum spread $S_D^{\max}$ is defined as the maximum achievable spread over the set I_Π of all possible interleavers:

$$S_D^{\max} = \max_{\Pi \in I_\Pi} S_D \,. \quad \textbf{(B-3)}$$

B.2 Upper bound of the maximum spread

The sphere packing approach allows to derive an upper bound of the maximum spread $S_D^{\max}$. The general idea is quite simple: for each of the N points of a given interleaver, we define a sphere of radius r. We show that, under a given hypothesis, the maximum radius so that the spheres are disjoint is related to the spread S_D of this interleaver. The maximum spread is then given by the maximum radius so that the sphere are all disjoint. This problem is not trivial, nevertheless, a volume argument can be used to bound $S_D^{\max}$. In fact, the total volume of the N spheres is at most equal to the total available volume of the space, this gives us a maximum radius S_D^{SB} called the sphere bound.

Let $F = [0, N-1[$ be the flat torus of the real numbers modulo N and let $\tilde{d}_1$ be the application from $F^D \times F^D$ to $\mathbb{R}^+$ which is defined as:

$$\forall X,Y \in F^D \times F^D, \quad \tilde{d}_1(X,Y) = \sum_{i=0}^{D-1} \left|x_i - y_i\right|_N \,. \quad \textbf{(B-4)}$$

One can note that $\tilde{d}_1$ is a distance over F^D. Indeed, $\forall X,Y \in \left(F^D\right)^2$ $\tilde{d}_1(X,Y) = \tilde{d}_1(Y,X)$, $\forall X,Y \in \left(F^D\right)^2$, $\tilde{d}_1(X,Y) = 0 \Leftrightarrow X = Y$ and $\forall X,Y,Z \in \left(F^D\right)^3$, $\tilde{d}_1(X,Z) \leq \tilde{d}_1(X,Y) + \tilde{d}_1(Y,Z)$. The proof of these properties is straightforward.

By definition of $\tilde{d}_1$, the spread S_D is also a distance over $E \times E$ and $S_D(k_1,k_2) = \tilde{d}_1(\Phi(k_1),\Phi(k_2))$.

Let us define d_1, the distance derived from the ℓ_1-norm, as $d_1(X,Y) = \sum_{i=0}^{D-1} \left|x_i - y_i\right| = \left\|X - Y\right\|_1$.

Lemma B-1: Let X and Y be two elements of $\mathbb{R}^D$. If $d_1(X,Y) \leq N/2$, then $\tilde{d}_1(X,Y) = d_1(X,Y)$.

Proof: If $d_1(X,Y) \leq N/2$, then for all $i \in \{0,...,D-1\}$, $\left|x_i - y_i\right| \leq N/2$ and thus, $\min\left(\left|x_i - y_i\right|, N - \left|x_i - y_i\right|\right) = \left|x_i - y_i\right|$.

Thus, the distance $\tilde{d}_1$ behaves locally as the distance d_1. Considering the sphere $\Sigma_D(A,r,\tilde{d}_1)$ with respect to the distance $\tilde{d}_1$, the centre *A* and the radius $r \le N/2$ in a *D*-dimensional space $\mathbb{R}^D$ and applying Lemma B-1, we obtain:

$$\Sigma_D(A,r,\tilde{d}_1) = \left\{B \in F^D / d_1(A,B) < r\right\}. \quad \textbf{(B-5)}$$

Hypothesis B-1: The value of *r* is below of equal to *N*/2 ($r \le N/2$).

According to hypothesis 1, the two distances $\tilde{d}_1$ and d_1 are equivalent. The notation of the sphere $\Sigma_D(A,r,\tilde{d}_1)$ is then simplified to $\Sigma_D(A,r)$. Note that this sphere is a square in dimension 2 and a regular octahedron in dimension 3, as shown in [Fedorov-53] and depicted in Figure B-1.

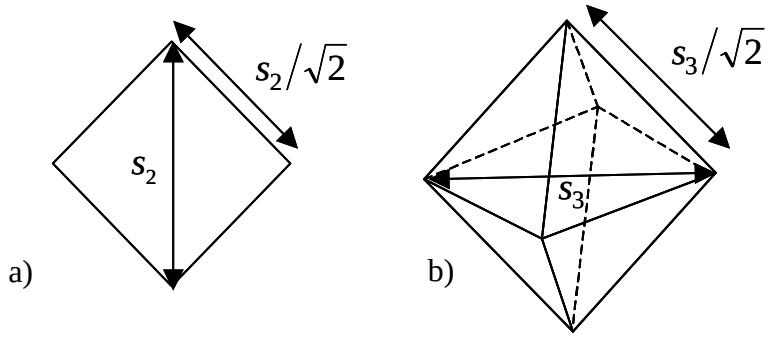


Figure B-1: Representation of the sphere $\Sigma_D(A,r)$: a) a square in dimension 2, b) an octahedron in dimension 3

Lemma B-2: Let S_D be the spread of an interleaver Π. Then:

$$\forall k_1, k_2 \in E^2, k_1 \ne k_2 \Rightarrow \Sigma_D(\Phi(k_1), S_D/2) \cap \Sigma_D(\Phi(k_2), S_D/2) = \varnothing. \quad \textbf{(B-6)}$$

Proof: If the intersection between the two spheres is not empty, there is a point *X* that verifies $\tilde{d}_1(\Phi(k_1), X) < S_D/2$ and $\tilde{d}_1(X, \Phi(k_2)) < S_D/2$. Since $\tilde{d}_1(\Phi(k_1), \Phi(k_2)) \le \tilde{d}_1(\Phi(k_1), X) + \tilde{d}_1(X, \Phi(k_2))$, then $\tilde{d}_1(\Phi(k_1), \Phi(k_2)) < S_D$ which is in contradiction with the definition 3 of the spread of an interleaver.

From the lemma 2, we deduce that the sum of the volume of the *N* spheres $\Sigma_D\left(\Phi(k), S_D/2\right)_{k \in E}$ is less or equal to the total space volume N^D. The value S_D^{SB}, called the SB, leading to equality, is thus an upper bound of $S_D^{\max}$.

B.3 Maximal spread for SSTCs

From Lemma B-2, the maximum spread value $S_2^{\max}$ of a 2-dimensional interleaver can be bounded by the SB S_2^{SB}. In fact, if $S_2^{SB}/2$ is less or equal to *N*/2 (hypothesis B-1), the area of the sphere $\Sigma_2(A, S_2^{SB}/2)$ is $\left(S_2^{SB}\right)^2/2$. Thus the SB leads to the following relation:

$$N \cdot \left(S_2^{SB}\right)^2 / 2 = N^2 \quad \Rightarrow \quad S_2^{SB} = \sqrt{2 \cdot N}\,. \tag{B-7}$$

The hypothesis $S_2^{SB} \le N$ is valid if $N \ge 2$. This is verified in all practical applications.

This SB can be achieved with deterministic interleavers. For example, if $N=2n^2$, where n is a positive integer, the maximum spread is $S_2^{\max}=2n$. This spread is obtained using simply the regular permutation defined as $\Pi_1(k) = (2n\text{-}1)\cdot k \bmod N$ [6]. Figure B-2 shows an example of interleaver with $n = 2$ and $N = 8$, *i.e.*, an interleaver of equation $\Pi_1(k) = 3k \bmod 8$. Hence, in the two-dimensional case, the upper bound on the maximum spread is the maximum achievable spread and $S_2^{\max} = S_2^{SB} = \sqrt{2N}$.

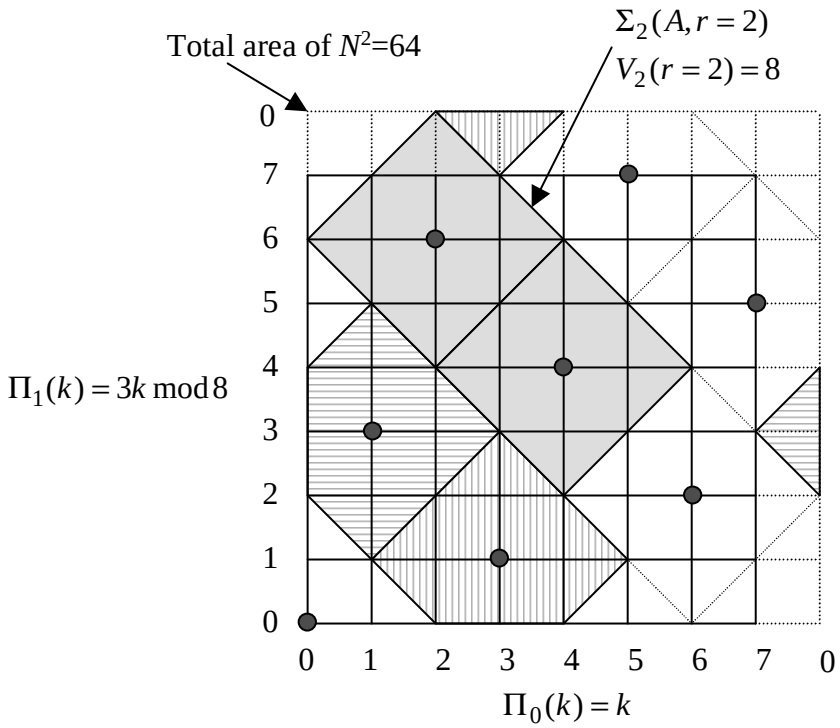


Figure B-2: Spread in the 2-dimensional case for $N = 8$, $S_2^{\max} = S_2^{SB} = 4$

B.4 Generalization to MSTCs

Let us consider a two-dimensional MSTC of parameters (N, M, K) associated to an interleaver that spreads uniformly the symbol of one slice in the natural order to the slices of the interleaver order. To this end, we use a spatial permutation bijective and K-periodic as described in chapter 3.3. In this case, let $\rho = M / K$ denote the number of symbols that are in the same slice in the natural and the interleaved order. Unlike with conventional SSTCs, for MSTCs, the flat torus representing the interleaver space is not unique. There are K^2 flat toruses of size M^2, one for each intersection between two slices of the natural and interleaved order. Thus, the sphere corresponding to the interleaver points loops in a single torus, in which the above-developed volume argument is used.

In the first step, we study the case where ρ is an integer greater than or equal to 2. Then we will generalize to the case where ρ is lower than or equal 1.

B.4.1 Case $\rho \geq 2$

The ρ symbols that belong to the same slice in the two dimensions of the code verify the above-discribed volume argument. Each of the ρ symbols is associated with a sphere of radius $S_2^{SB}/2$. The ρ spheres fills totally the space of the flat torus (of area M^2) corresponding to the two slices. Therefore, this condition leads to the following relation:

$$\rho \cdot \left(S_2^{SB}\right)^2 / 2 = M^2 . \quad \textbf{(B-8)}$$

Replacing the expression of ρ in (B-8) and using $M = N / M$ yields:

$$S_2^{SB} = \sqrt{2 \cdot N} . \quad \textbf{(B-8)}$$

The maximal spread of a two dimensional MSTC with $\rho \geq 2$ is equal to the sphere bound of a conventional SSTC: $S_2^{\max} = S_2^{SB} = \sqrt{2N}$. As for SSTC, this bound is achieved when the spheres fill totally the space. Interleavers based on regular permutation can be used to achieve this objective.

B.4.2 Case $\rho \leq 1$

When $\rho \leq 1$, and the symbols of a single slice are distributed over distinct slices in the other dimension, there exists only a single symbol in each flat torus, which can fill totally the space. However, as shown in Figure B-3, the radius of the sphere must remain lower than $M/2$, such that the sphere does not overlap on itself. The extreme case of this condition leads to $S_2^{SB}/2 = M$, which gives a sphere bound of $2M$. Thus, the maximal spread of a two-dimension MSTC with $\rho \leq 1$ is equal to $S_2^{\max} = 2M$. This result is important, because it shows that the maximal spread is not a function of the frame size N, but depends on the size of the slice M. Consequently, the slice size of a MSTC must be chosen so that it guarantees a minimal spread.

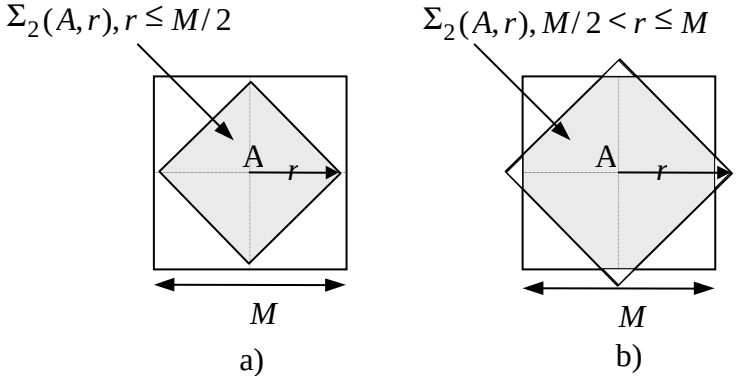


Figure B-3: Sphere $\Sigma_D(A, r, \tilde{d}_1)$ in a 2-dimensional space in the case: a) $r \leq M/2$ and b) $M/2 < r \leq M$

Appendix C

Performance of Multiple Slice Turbo Codes

This appendix describes other performance curves of MSTCs using a hierarchical interleaver design using the optimization steps developed in chapter 3.

Appendix C. Performance of Multiple Slice Turbo Codes

Performance of MSTCs with parameters (1632, 204, 8), (3264, 408, 8) and (6528, 816, 8) are presented in Figure C-4, Figure C-5 and Figure C-6 for code rates 1/2, 2/3 and 3/4, respectively. The interleavers were optimized using the approach developed in chapter 3.5. The spatial permutation Π_S for the three codes is a circular shift of amplitude $A = \{0, 2, 1, 5, 4, 7, 3, 6\}$. The temporal permutation is an ARP interleaver using the parameters given in Table C-1. The same interleaver parameters are used for all code rates. The curves are obtained with 8 iterations of the Log-MAP algorithm with $L = 51$ and $w_d = 4$ bits.

	λ	μ(t mod 4)
(1632, 204, 8)	11	{0, 76, 0, 108}
(3264, 408, 8)	37	{0, 106, 80, 2}
(6528, 816, 8)	23	{0, 0, 124, 48}

Table C-1: Interleaver parameters for MSTCs (1632, 204, 8), (3264, 408, 8) and (6528, 816, 8)

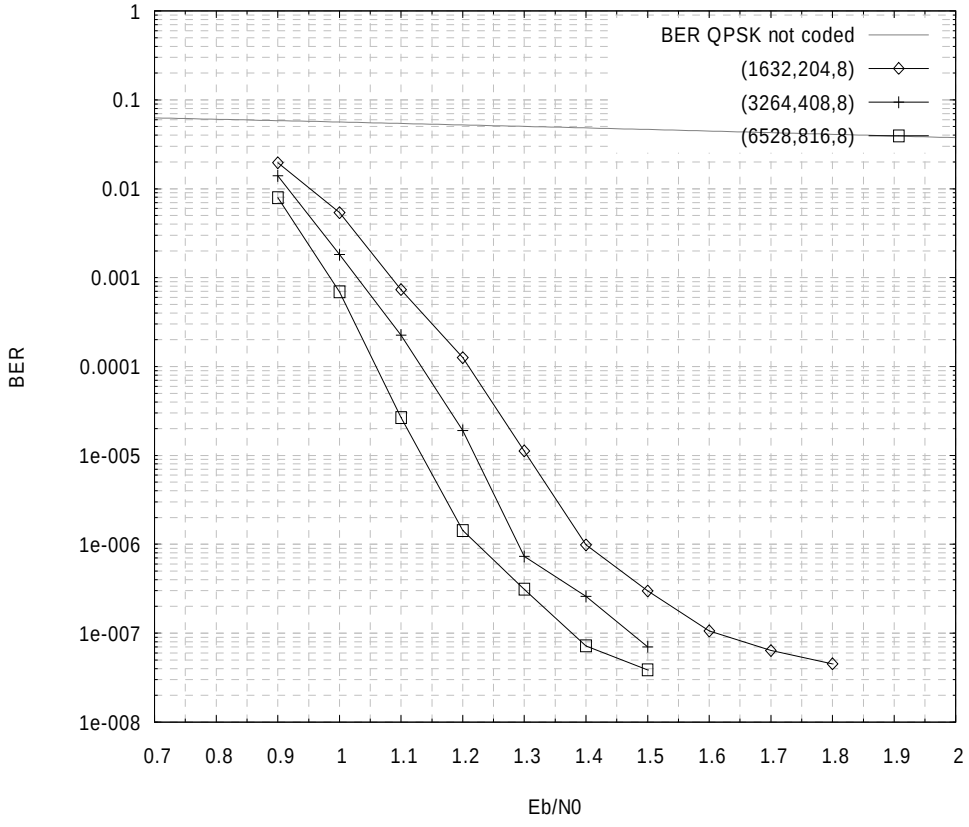


Figure C-4: BER performance of rate 1/2 (1632, 204, 8), (3264, 408, 8) and (6528, 816, 8) MSTCs

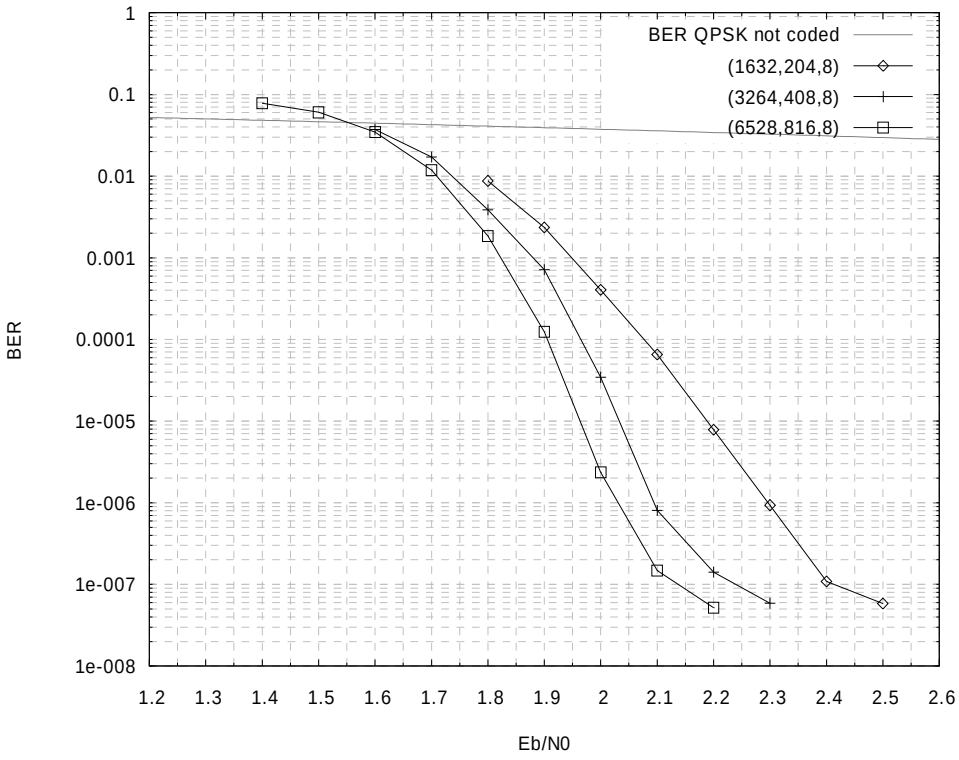


Figure C-5: BER performance of rate 2/3 (1632, 204, 8), (3264, 408, 8) and (6528, 816, 8) MSTCs

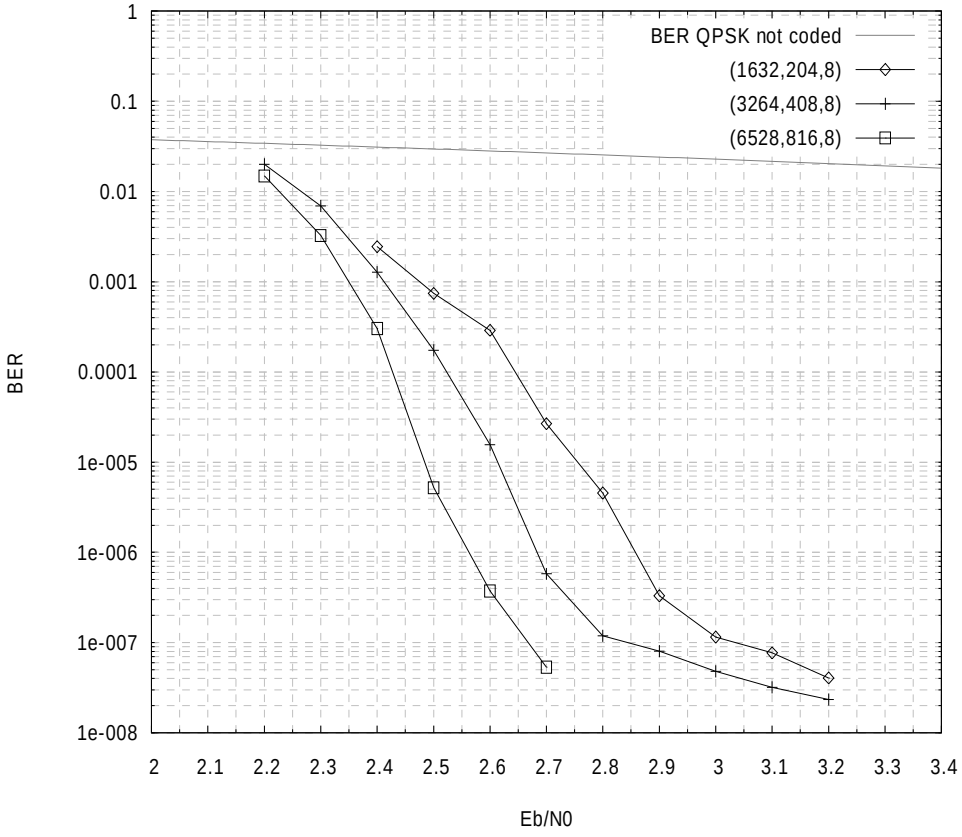


Figure C-6: BER performance of rate 3/4 (1632, 204, 8), (3264, 408, 8) and (6528, 816, 8) MSTCs

Appendix D

Design of Three-Dimensional Multiple Slice Turbo Codes

This appendix reproduces the paper published in [Gnaedig et al-05b], which extends the MSTCs construction to three-dimensional multiple turbo codes.

EURASIP Journal on Applied Signal Processing 2005:6, 808–819
© 2005 David Gnaedig et al.

Design of Three-Dimensional Multiple Slice Turbo Codes

David Gnaedig

TurboConcept, Technopôle Brest-Iroise, 115 rue Claude Chappe, 29280 Plouzané, France
Email: david.gnaedig@turboconcept.com

LESTER, Université de Bretagne-Sud, BP 92116, 56321 Lorient Cedex, France

ENST Bretagne, Technopôle Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France

Emmanuel Boutillon

LESTER, Université de Bretagne-Sud, BP 92116, 56321 Lorient Cedex, France
Email: emmanuel.boutillon@univ-ubs.fr

Michel Jézéquel

ENST Bretagne, Technopôle Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France
Email: michel.jezequel@enst-bretagne.fr

Received 8 October 2003; Revised 8 November 2004

This paper proposes a new approach to designing low-complexity high-speed turbo codes for very low frame error rate applications. The key idea is to adapt and optimize the technique of multiple turbo codes to obtain the required frame error rate combined with a family of turbo codes, called multiple slice turbo codes (MSTCs), which allows high throughput at low hardware complexity. The proposed coding scheme is based on a versatile three-dimensional multiple slice turbo code (3D-MSTC) using duobinary trellises. Simple deterministic interleavers are used for the sake of hardware simplicity. A new heuristic optimization method of the interleavers is described, leading to excellent performance. Moreover, by a novel asymmetric puncturing pattern, we show that convergence can be traded off against minimum distance (i.e., error floor) in order to adapt the performance of the 3D-MSTC to the requirement of the application. Based on this asymmetry of the puncturing pattern, two new adapted iterative decoding structures are proposed. Their performance and associated decoder complexities are compared to an 8-state and a 16-state duobinary 2D-MSTC. For a 4 kb information frame, the 8-state trellis 3D-MSTC proposed achieves a throughput of 100 Mbps for an estimated area of 2.9 mm^2 in a 0.13 μm technology. The simulation results show that the FER is below 10^{-6} at SNR of 1.45 dB, which represents a gain of more than 0.5 dB over an 8-state 2D-MSTC. The union bound gives an error floor that appears at FER below 10^{-8}. The performance of the proposed 3D-MSTC for low FERs outperforms the performance of a 16-state 2D-MSTC with 20% less complexity.

Keywords and phrases: turbo codes, interleavers, multiple turbo codes, tail-biting codes, slice turbo codes.

1. INTRODUCTION

Turbo codes [1] are known to be very close to the Shannon limit. They are often constructed as a parallel concatenation of binary or duobinary [2] 8-state or 16-state recursive systematic convolutional codes. Turbo codes with 8-state trellises have a fast convergence at low signal-to-noise ratios (SNRs) but an error floor appears at high SNRs due to the weak minimum distance of these codes. For interactive, low-latency applications such as video conferencing requiring a very low frame error rate, an automatic repeat request (ARQ) system combined with a turbo code [3] cannot be used. Since this kind of application requires low latency, the block size cannot exceed few thousand bits. At constant block size, for very low frame error rate applications, several alternatives can be used. First, the more efficient 16-state trellis encoder can replace the 8-state trellis encoder at a cost of a double hardware complexity [4]. These encoders have the same waterfall region but the error floor region is considerably lowered due to the higher minimum distance. The second

This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

alternative is to use a serial concatenation with an outer code, for example, either a Reed Solomon code [5] or a BCH code [6]. To achieve very good performance with these concatenations, a very large interleaver is needed to uniformly spread the errors at the output of the turbo decoder. The use of such an interleaver results in a long latency that is not acceptable for interactive services. Moreover, these serial concatenations decrease spectral efficiency.

The third alternative is to use multiple turbo codes. multiple turbo codes were introduced in [7] by adding a third dimension to the two-dimensional turbo code using reduced state trellises. It was shown in [8] that an increase of 50% of the minimum distance can be obtained by adding a third dimension to the turbo code. Using an 8-state trellis, this parallel code construction results in an equivalent or a higher minimum distance than for 16-state two-dimensional turbo codes. Other work on the constituent codes of three-dimensional turbo codes has been done by analyzing their convergence properties using extrinsic information transfer analysis [9], but these analyses do not handle the problem of designing good three-dimensional interleavers. Most of the designs of multiple turbo codes use random or *S*-random interleavers [7, 10], which are not efficient for scalable hardware implementation. Indeed, these types of interleavers are implemented in hardware as a table containing the interleaved address of all the symbols of the frame. Since practical applications generally require versatility, that is, several frame lengths and code rates, the storage of all the possible interleavers can represent a huge amount of memory.

In this paper, we generalize the multiple slice turbo codes (MSTCs) presented in [11] to the three-dimensional case. The idea is to propose a new and more efficient coding solution for high throughput, low hardware complexity, very low frame error rate applications. The use of MSTCs guarantees a parallel decoding architecture thanks to the way they are constructed. The careful design of a deterministic three-dimensional interleaver leads to a very simple address generation scheme and a high minimum distance for the code.

The paper is divided into five sections. In Section 2, multiple slice turbo codes are described, together with the interleaver construction. In Section 3, the design of three-dimensional MSTCs and of the interleavers is addressed. Then new decoding structures for three-dimensional codes are introduced in Section 4. Finally, the performance of the 2D-MSTC and 3D-MSTC is summarized in Section 5 and their complexities are compared in Section 6.

2. MULTIPLE SLICE TURBO CODES

The idea of multiple slice turbo codes (MSTC) was proposed by Gnaedig et al. [11]. The same idea has been mentioned independently in [12]. The aim of MSTCs is to increase by a factor P (the number of slices) the decoding parallelism of the turbo decoder without memory duplication. The principle of MSTCs is based on the following two properties.

(i) In each encoding dimension, the information frame of the N m-binary symbols is divided into P blocks (called "slices") of M symbols, where $N = M \cdot P$. Then, each slice is encoded independently by a convolutional recursive systematic convolution (CRSC) code. Finally, puncturing is applied to generate the desired code rate.

(ii) The permutation Π^i between the natural order of the information frame and the interleaved order of the ith dimension has a particular structure avoiding memory conflicts when a parallel architecture with P decoders is used.

The resulting MSTC is represented by the triplet (N, M, P).

After describing the construction of the interleaver, we will recall some simple rules for building efficient two-dimensional MSTCs (2D-MSTCs). Then, we will generalize these rules to the three-dimensional case (3D-MSTC). Note that all the results given in this paper are obtained with duobinary turbo codes [2]. The same results can also be used for classical turbo codes.

2.1. Multiple slice interleaver construction

The interleaver is designed jointly with the memory organization to allow parallel decoding of the P slices. In other words, at each symbol cycle k, the interleaver structure allows the P decoders to read and write the P necessary data symbols from the P memory banks $\mathrm{MB}_0, \mathrm{MB}_1, \ldots, \mathrm{MB}_{P-1}$ without conflict. Since only one read can be made at any given time from a single-port memory, in order to access P data symbols in parallel, P memory banks are necessary.

The interleaver design is based on the one proposed in [13]: The interleaver structure presented in Figure 1 is mapped onto a hardware decoding architecture allowing a parallel decoding process.

The frame is first stored in the natural order in P memory banks, that is, the symbol with index j is stored in the memory bank $\lfloor j/M \rfloor$ at the address $j \bmod M$.

When considering the encoding (or decoding) of the ith dimension of the turbo code, the encoding (decoding) process is performed on independent consecutive blocks of M symbols of the permuted frame: the symbol with index k is used in slice $r = \lfloor k/M \rfloor$ at temporal index $t = k \bmod M$. Note that $k = M \cdot r + t$, where $r \in \{0, \ldots, P-1\}$ and $t \in \{0, \ldots, M-1\}$. For the symbol with index k of the interleaved order, the permutation Π^i associates a corresponding symbol in the natural order with index $\Pi^i(k) = \Pi^i(t, r)$. To avoid memory conflict, the interleaver function is split into two levels: a spatial permutation $\Pi^i_S(t, r)$ and a temporal permutation $\Pi^i_T(t, r)$, as defined in the following:

$$\Pi^i(k) = \Pi^i(t, r) = \Pi^i_S(t, r) \cdot M + \Pi^i_T(t, r). \tag{1}$$

The symbol with index k in the interleaved order is read from memory bank $\Pi^i_S(t, s)$ at address $\Pi^i_T(t, r)$. When coding (or decoding) the noninterleaved dimension (or dimension 0), the frame is processed in the natural order. The spatial and temporal permutations are then simply replaced by *identity* functions ($\Pi^0_S(t, r) = r$ and $\Pi^0_T(t, r) = t$).

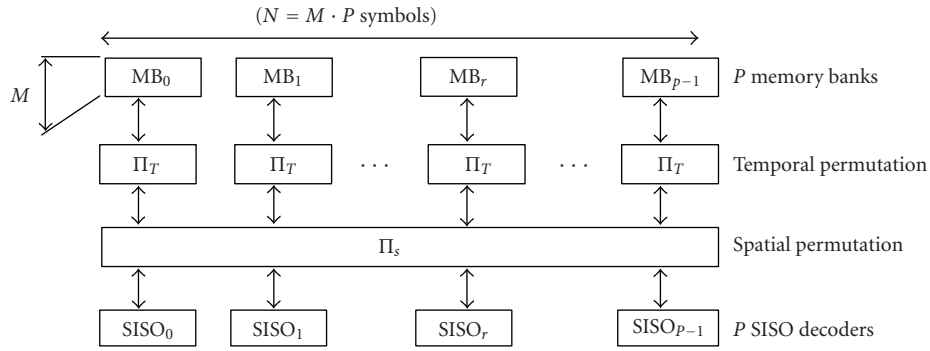


Figure 1: Interleaver structure for the (N, M, P) code.

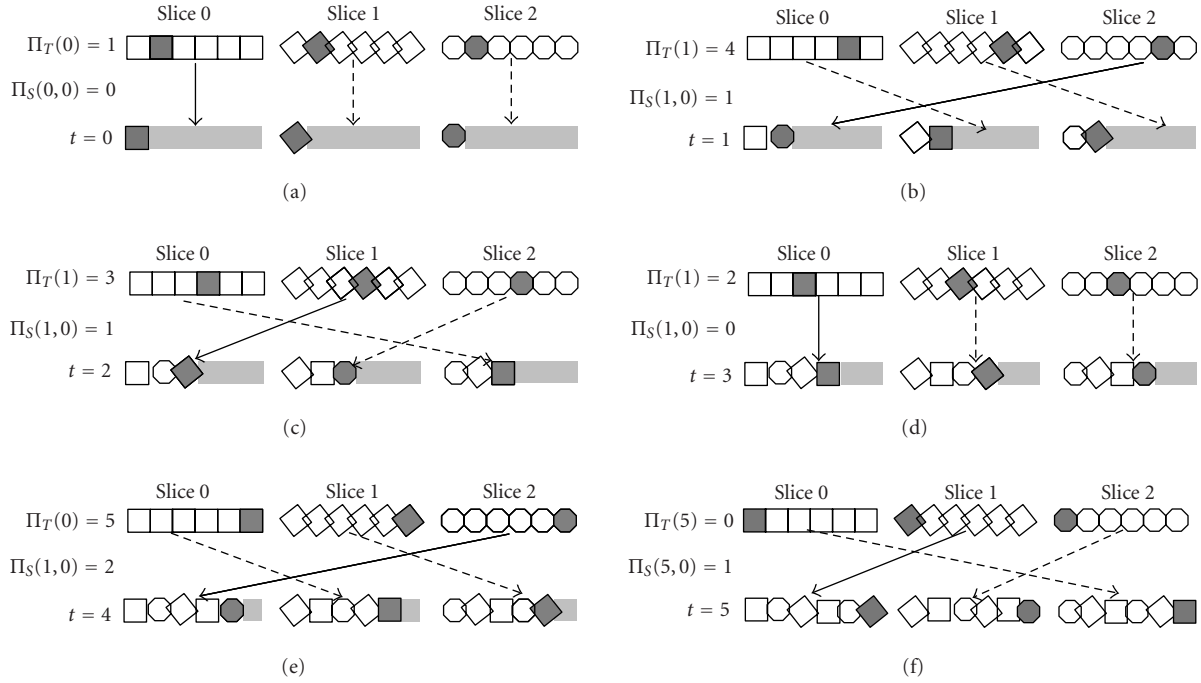


Figure 2: A basic example of an $(18, 6, 3)$ code with $\Pi_T(t) = \{1, 4, 3, 2, 5, 0\}$ and $A(t \bmod 3) = \{0, 2, 1\}$.

The spatial permutation allows the P data read-out to be transferred to the P decoders performing the max-log-MAP algorithm [14] (called the SISO algorithm). They are called SISO in Figure 1. Decoder r receives the data from memory bank $\Pi_S^i(t, s)$ at time t. For all fixed t, the function $\Pi_S^i(t, r)$ is then a bijection between the decoder index $r \in \{0, \ldots, P-1\}$ and the memory banks $\{0, \ldots, P-1\}$. To simplify the design, the shuffle network Π_S of Figure 1 is made with a simple barrel shifter, that is, for any given time t, $\Pi_S^i(t, r)$ is a rotation of amplitude $A^i(t)$. Furthermore, to maximize the shuffling between dimensions, we constrain function $\Pi_S^i(t, r)$ such that every P consecutive symbols of any slice come from P distinct memory banks. Thus, for a given r, the function $\Pi_S^i(t, r)$ is bijective and P-periodic. This means that for a given r, the function $\Pi_S^i(t, r)$ is a bijection between the temporal index $t \in \{0, \ldots, P-1\}$ and the set $\{0, \ldots, P-1\}$ of memory bank indices. Moreover, $\Pi_S^i(t, r)$ should also be P-periodic in the variable t, that is, $\Pi_S^i(t, r) = \Pi_S^i(t + P, r)$. The amplitude of the rotation $A^i(t)$ is then a P-periodic function and the spatial permutation is given by

$$\Pi_S^i(t, s) = \left(A^i(t \bmod P) + s\right) \bmod P. \tag{2}$$

2.2. A simple example of an interleaver

We construct a simple $(N, M, P) = (18, 6, 3)$ 2D-MSTC to clarify the interleaver construction. Let the temporal permutation be $\Pi_T(t) = \{1, 4, 3, 2, 5, 0\}$ (i.e., $\Pi_T(0) = 1$, $\Pi_T(1) = 4, \ldots$) and let the spatial permutation be a circular shift of amplitude $A(t \bmod 3)$, that is, the slice of index r is associated with the memory bank of index $\Pi_S(t, r) = (A(t \bmod 3) + r) \bmod 3$, with $A(t \bmod 3) = \{0, 2, 1\}$. The spatial permutation is then bijective and 3-periodic.

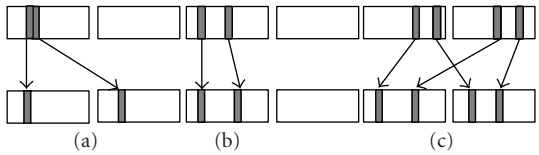


Figure 3: Primary and secondary cycles: (a) noncycle, (b) primary cycle, and (c) secondary cycle.

The interleaver is illustrated in Figure 2, which shows the permutations for the 6 temporal indices $t = 0$(2.a), $t = 1$(2.b), $t = 2$(2.c), $t = 3$(2.d), $t = 4$(2.d), and $t = 5$(2.c). The 18 symbols in the natural order are separated into 3 slices of 6 symbols corresponding to the first dimension. In the second dimension, at temporal index t, symbols $\Pi_T(t)$ are selected from the 3 slices of the first dimension, and then permuted by the spatial permutation $\Pi_S(t, r)$. For example, at temporal index $t = 1$(b), symbols at index $\Pi_T(1) = 4$ are selected. Then, they are shifted to the left with an amplitude $A(1 \bmod 3) = 2$. Thus, symbols 4 from slices 0, 1, and 2 of the first dimension go to slices 1, 2, and 0 of the second dimension, respectively.

2.3. Optimization of a two-dimensional interleaver

Optimization of an interleaver Π aims to fulfill two performance criteria: first, a good minimum distance for the asymptotic performance of the code at high signal-to-noise ratios (SNRs); second, fast convergence, that is, to obtain most of the coding gain performance in few decoding iterations at low SNRs. The convergence is influenced by the correlation between the extrinsic information, caused by the presence of short cycles in the interleaver. The cycles that are more likely to occur are primary and secondary cycles, as depicted in Figure 3. When these cycles correspond to combinations of low input-weight patterns leading to low weight codewords in the RSC constituent codes, they are called primary and secondary error patterns (PEPs and SEPs).

In [11, 15], the influence of the temporal and spatial permutations on these error patterns has been studied, using the following:

$$\Pi_T(t) = \alpha \cdot t \bmod M, \tag{3}$$

$$\Pi_S(t, s) = A(t \bmod P) + s \bmod P, \tag{4}$$

where α and M are mutually prime, and $A(t \bmod P)$ is a bijection between $\{0, \ldots, P-1\}$ and $\{0, \ldots, P-1\}$. Equation (4) for the spatial permutation is a circular shift of amplitude $A(t \bmod P)$, which can be easily implemented in hardware.

In order to characterize primary cycles and PEPs, other authors have introduced spread [16, 17, 18] and used the spread definition to improve the interleaver gain. In [11], an appropriate definition of spread is used taking into account the slicing of the constituent code. The spread between two symbols is defined as $S(k_1, k_2) = |k_1 - k_2|_M + |\Pi(k_1) - \Pi(k_2)|_M$, where $|a - b|_C$ is equal to $\min(|a - b|, C - |a - b|)$ if $\lfloor a/C \rfloor = \lfloor b/C \rfloor$ (this condition implies that the symbols a and b belong to the same slice when $C = M$), and is equal to infinity otherwise. The overall minimum spread is then defined as $S = \min_{k_1,k_2}[S(k_1, k_2)]$. Low weight PEPs are eliminated with high spread. Since the spatial permutation is P-periodic and bijective, two symbols separated by less than P symbols in the interleaved order are not in the same slice in the natural order. Their spread is then infinite. Using the definition of spread, the optimal parameter α maximizes the spread of the symbols separated by exactly P symbols.

Since the weight of the SEPs does not increase with high spread, we choose the spatial permutation in order to maximize the weight of these patterns. This weight is maximized for irregular spatial permutations. For a regular spatial permutation (e.g., $A(t) = a \cdot t + b \bmod P$, where a and P are relatively prime and $b > 0$) many SEPs with low Hamming weight are obtained [11]. To characterize the irregularity of a permutation, dispersion was introduced in [17]. In [15], an appropriate definition of dispersion is proposed to characterize the irregularity of the spatial permutation. First, for a couple $t_1, t_2 \in \{0, \ldots, P-1\}^2$, a displacement vector $D_v(t_1, t_2)$ of the spatial permutation is defined as $D_v(t_1, t_2) = (\Delta t, \Delta A)$, where $\Delta t = |t_1 - t_2|M$ and $\Delta A = |A(t_1) - A(t_2)|_P$. Let $D = \{t_1, t_2 \in \{0, \ldots, P-1\}^2, D_v(t1, t2)\}$ be the set of displacement vectors. The dispersion is then defined as the cardinality of D, that is, the number of different couples. It can be observed that the number of low weight SEPs decreases with high dispersion. This simple property is explained in detail in [15]. Some other criteria about the choice of the spatial permutation are also given in [15].

The criteria of spread and dispersion maximization increase the weight of PEPs and SEPs and improve the convergence of the code. But, with increasing frame size, the study of PEPs and SEPs alone is not sufficient to obtain efficient interleavers. Indeed, more complex error patterns appear, penalizing the minimum distance. In practice, the analysis and the thorough counting of these new patterns are too complex to be performed. Thus, to increase the minimum distance of the code, four coefficients $\beta(i)_{i=0,\ldots,3}$, multiple of 4, are added to the temporal permutation:

$$\Pi_T(t) = \alpha \cdot t + \beta(t \bmod 4) \bmod M. \tag{5}$$

The minimum distance is evaluated using the error impulse method proposed by Berrou et al. [19], which gives a good approximation of the minimum distance. Its results can be used to compute the union bound of the code.

3. THREE-DIMENSIONAL MULTIPLE SLICE TURBO CODES

In order to lower the error floor of the two-dimensional 8-state MSTC, a third dimension is introduced into the code. The goal is to increase the weight of the low weight error patterns of the 2D-MSTC, while maintaining good convergence at low SNRs. The interleaver of the third dimension has the same structure as the interleaver of 2.1 in order to allow the parallel decoding of the slices in each of the three dimensions.

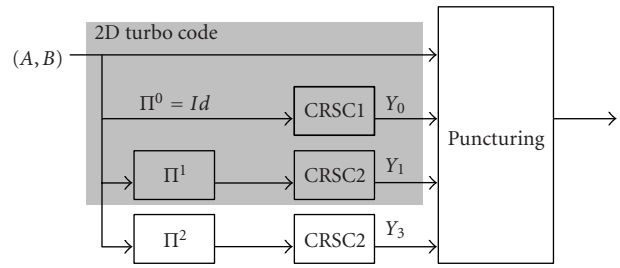


Figure 4: Three-dimensional multiple slice turbo code structure.

3.1. 3D slice turbo code construction

The generalization of the 2D slice turbo code to a 3D slice turbo code is straightforward (see Figure 4). Like for the second dimension, the third dimension is also sliced and its associated interleaver Π^2 has the same structure as that of Π^1:

$$\Pi^2(k) = \Pi^2(t,r) = \Pi^2_S(t,r) \cdot M + \Pi^2_T(t,r). \quad (6)$$

The generation of the addresses in the third dimension is simple and, due to the construction of Π^2, the architecture of Figure 1 can also compute the third dimension interleaver, with a degree P of parallelism, with negligible extra hardware (only interleaver parameters need to be stored).

Since the first initial paper on turbo codes in 1993 [1], much work has been done on efficient design methodologies for obtaining good 2D interleavers [16, 20, 21]. There are several papers dealing with multidimensional turbocodes [7, 9, 22, 23] but, unfortunately, very few papers consider the complex problem of the construction of good 3D interleavers. The 3D interleaver designs presented in [7, 10, 24] are based on random and S-random interleavers. Note that dithered relative prime (DRP) interleavers introduced by Crozier and Guinand in [25] have been used in [26] to design low-complexity multiple turbo codes. Moreover, none of these papers deals with the construction of good interleavers for duobinary codes. We have restricted our efforts to obtaining a class of couples (Π^1, Π^2) for 3D-MSTC verifying the following three properties.

(1) A 3D interleaver (Id, Π^1, Π^2) is said to be a "good" 3D interleaver if the three 2D interleavers defined by $(Id, \Pi^1), (Id, \Pi^2), (\Pi^1, \Pi^2) = (Id, \Pi^{1^{-1}} \circ \Pi^2)$ are "not weak," that is, the spreads for the temporal permutations and the dispersions for the spatial permutations are optimized by using the 2D interleaver construction developed in Section 2.3.

(2) There is a maximum global spreading of the message symbols over the slices, that is, the maximum number S_3 of common symbols between 3 distinct slices should be minimized.[1]

[1] The minimal value of S_3 is $\lfloor M/P^2 \rfloor$. When P^2 divides M, the appropriate choice of temporal permutation leads to $S_3 = M/P^2$.

Table 1: Puncturing patterns $P1$, $P2$, and $P3$ of different periods.

h	$h=1$	$h=3$	$h=4$	$h=8$	$h=16$
$P1$	1	011	0111	01111111	0111111111111111
$P2$	1	101	1101	11110111	1111111011111111
$P3$	0	110	0101	10001000	10000000100000000

(3) There is a regular intersymbol permutation ((A, B) becomes (B, A)): in the second dimension, all even indices are permuted; in the third dimension, all odd indices are permuted.

Note that these three conditions are an *a priori* choice, based on the authors intuition and on their work on 2D interleavers. Simulation results show that they effectively lead to efficient 3D turbo codes. Since the constituent codes are duobinary codes of rate 2/3, the overall rate of the turbo code without puncturing is 2/5. Puncturing is applied on the parity bits and on the systematic bits to generate the desired code rate. It will be shown, however, in the sequel that the puncturing strategy has a dramatic influence on performance. Moreover, the interleaver optimization process can use the properties of the puncturing strategy, as will be seen in the next section.

3.2. Puncturing

With irregular spatial rotations, the influence of puncturing on the performance of the rate 1/2 three-dimensional multiple slice turbo code has been studied. For a duobinary 3D-MSTC, every incoming symbol (2 bits) generates three parity bits y_1, y_2, y_3. Thus, to obtain a rate 1/2, one third of the parity bits has to be punctured. We define the puncturing patterns $P1$, $P2$, and $P3$ of parity bits y_1, y_2, y_3 as a stream of bit of periodicity h, where a "1" means that the parity bit is transmitted and a "0" means that the parity bit is discarded. In our test, the puncturing is uniformly distributed among the different symbols, that is, one and only one parity bit is discarded for each information symbol.

Table 1 gives different puncturing patterns for $h = 1, 3, 4, 8$, and 16. The case $h = 1$ corresponds, in fact, to a standard 2D code. The case $h = 3$ is a regular three-dimensional code with uniform protection for the three dimensions. For $h = 4, 8, 16$ (h a power of 2), $P1$ is constructed with a "0" in first position, "1" elsewhere, $P2$ is constructed with a "0" in the $h/2 - 1$ position, "1" elsewhere, and finally $P3$ is constructed with a "1" in the first and the $h/2$ position and "0" elsewhere. This construction guarantees that a single parity bit is punctured for every incoming symbol.

The performance of different puncturing patterns of period $h = 1, 3, 4, 8, 16, 32$, and 64 is given in Figure 5 for a duobinary 8-state MSTC with parameters (2056, 256, 8). The decoding is a floating point max-log-MAP algorithm [14], with 10 decoding iterations. The iterative decoding method used by the decoder is the extended serial decoding structure proposed in [27]. More details on this method are given in Section 4.

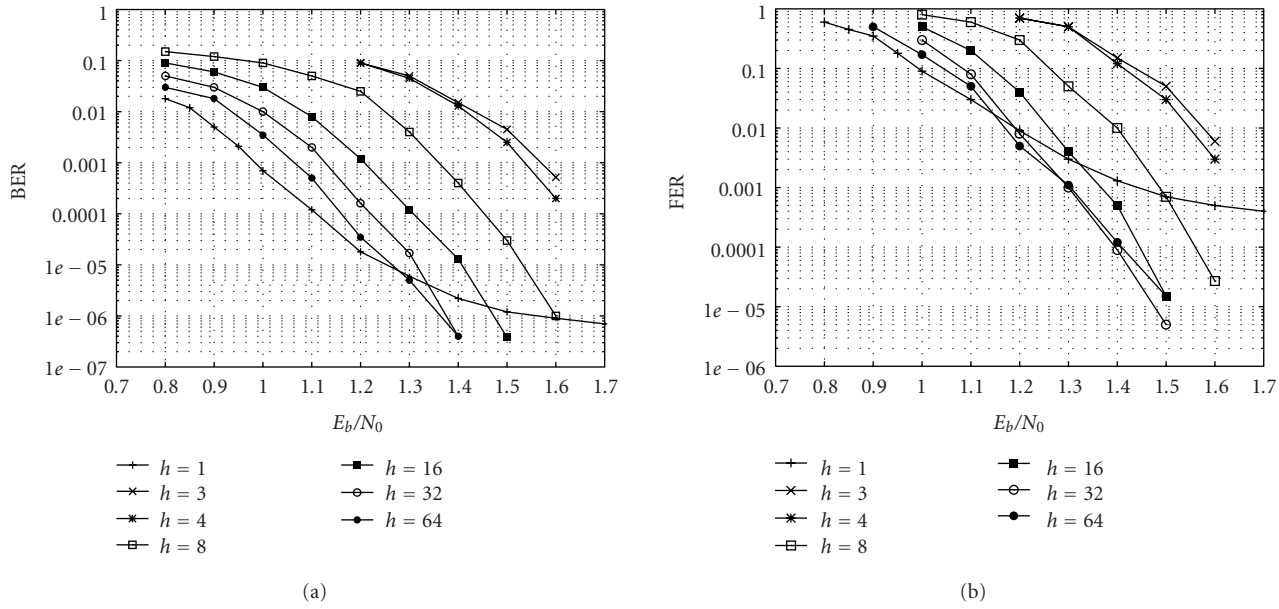


Figure 5: Performance of (2048, 256, 8) duobinary 8-state codes with different puncturing patterns with QPSK modulation on an AWGN channel.

The regular 3D code ($h = 3$) has no error floor due to its very high minimum distance: the impulsive estimation method [19] gives a minimum distance greater than 50. The drawback is a convergence loss of 0.5 dB at a bit error rate of 10^{-4} compared to the 2D code ($h = 1$). This loss of convergence has already been noted in the literature for binary turbo codes [8]. It can be explained by the fact that for every information symbol, the information relative to this symbol is spread over three trellises, instead of two for a 2D code. Hence, the waterfall region of the 3D code is at a considerably higher SNR than that of the 2D code.

As shown in [28], a high minimum distance is not needed to achieve a target FER of 10^{-7}. Indeed, for a 4 kb frame, the matched Hamming distance (MHD) is around 35. To reduce the minimum distance of the 3D code and to improve its convergence, we tend towards a 2D code by puncturing the third dimension more and more, while the first two dimensions are evenly and far more protected. The 3D codes with increasing puncturing period h given in Table 1 tend towards the 2D code: convergence at lower SNR and lower minimum distance. The 3D code is thus designed to trade off the loss of convergence with a minimum distance close to the MHD. The 3D code with puncturing period $h = 64$ has a convergence loss of less than 0.1 dB, but an error floor appears. The code of puncturing period $h = 32$ seems to have a reduced convergence loss and a high minimum distance.

The asymmetry in the protection of the three dimensions will be used in the decoder to reduce its complexity and improve its performance. Moreover, the optimal interleaver design depends on the puncturing patterns of the three dimensions. Thus, the interleaver optimization process has been redefined in order to take into account the asymmetric puncturing of the code: the first interleaver is chosen to obtain a good 2D-MSTC with the two highly protected dimensions. Then, the parameters of the second interleaver, that is, the third dimension, are selected in order to maximize the minimum distance of the code. The minimal distance is evaluated using the error impulse method [19]. This optimization process in two steps converges easily to an optimal solution and leads to an estimated minimum distance of 34, given by the error impulse method.

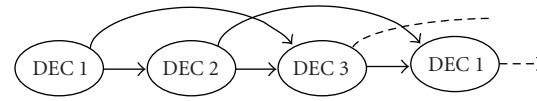


Figure 6: Extended serial decoding structure.

4. DECODING STRUCTURE

After describing the classical extended serial method [27], we study the impact of the scaling factors used to scale the extrinsic information during the decoding process [29]. We will show that an appropriate choice of scaling factor can increase performance while reducing decoding complexity (hybrid extended serial method). Then, we propose a suboptimal decoding method (partial serial method and hybrid partial serial method) that allows the third dimension to be decoded with a classical two-dimensional turbo decoder thanks to negligible additional hardware. Performance of the different coding schemes is also given.

4.1. Extended serial structure

The extended serial (ES) decoding structure is depicted in Figure 6. The three-dimensions are decoded sequentially,

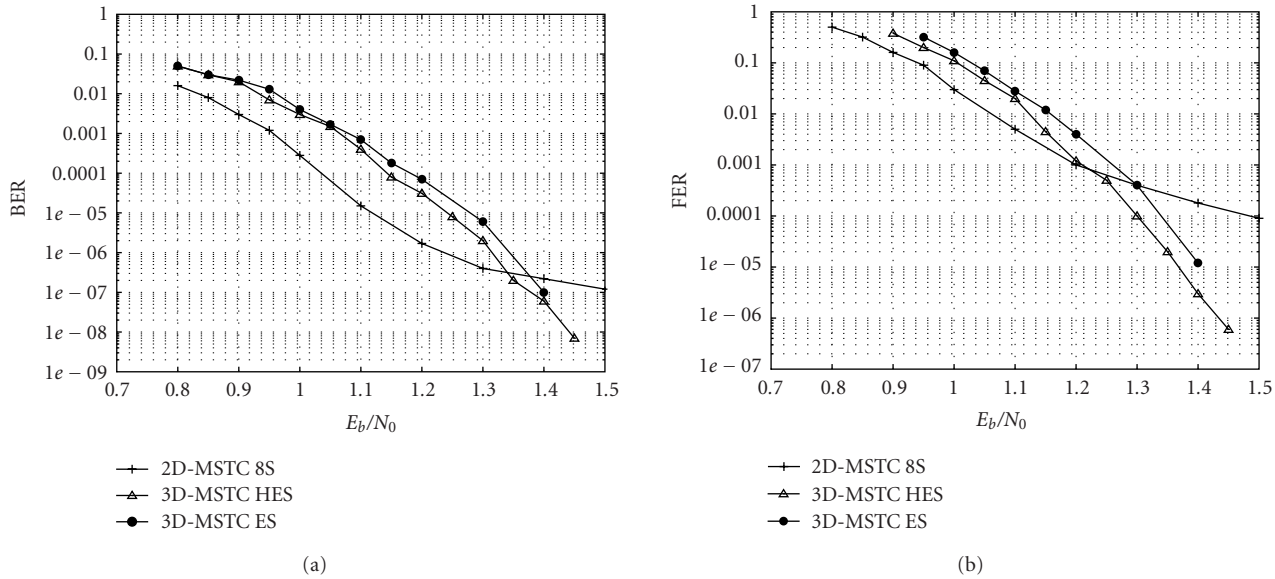


Figure 7: BER and FER comparison between the conventional ES structures and the hybrid HES structures for (2048, 256, 8) duobinary turbo codes of rate 1/2 over an AWGN channel with the max-log-MAP algorithm (10 full iterations) and $h = 32$.

Figure 8: Partial serial decoding structure and extrinsic memory content (E1, E2, and E3 correspond to extrinsic information produced by dimension 1, 2, and 3, respectively).

and each dimension receives information from the other two dimensions. This structure is repeated periodically with a period of 3.

With the ES architecture, each decoder uses the extrinsic information from the other two decoders, and therefore at least two extrinsic data per symbol must be stored. Thus the extrinsic memory is doubled.

4.2. Optimization of the decoding structure

Since the three-dimensional code is asymmetric and the third dimension is weak, during the first iterations, the reliability of the extrinsic information of the third dimension is low. In other words, during the first decoding iterations, the computation of the third dimension is useless and can thus be discarded. Moreover, for a max-log-MAP algorithm, to avoid performance degradation, extrinsic information is usually scaled by an SNR-independent scaling factor [29]. This scaling factor increases along the iterations.

By simulation, two different sets of scaling factors were jointly optimized, one for the first two equally protected dimensions and one for the weaker third dimension. For the third dimension, typically, the scaling factor is 0 for the first iterations (the third dimension is not decoded in practice) then it grows from 0.2 to 1 during the last iterations.

Thus, during the first iterations, the turbo decoder iterates only between the first two more protected dimensions (conventional two-dimensional serial structure S). Then, during the last iterations, an ES structure is used. This new structure will be called the hybrid extended serial (HES) structure.

Figure 7 compares the performance of the conventional nonoptimized ES structure with the performance of the optimized hybrid HES structure for 10 decoding iterations. For the hybrid structure, the third dimension is not decoded during the first 5 iterations. Then its scaling factor grows from 0.2 to 1 during the 5 last iterations. For the ES structure, the same scaling factor growing from 0.7 to 1 is used for the three dimensions. The simulation results show that the optimized structure slightly improves the performance for less computational complexity and negligible additional hardware. Unlike the classical structure, the optimized structure takes into account this unequal protection to improve decoding performance.

4.3. Partial serial structure

The ES decoder requires an additional extrinsic memory in order to store the extrinsic information from the last two decoding iterations. This extra memory leads to additional

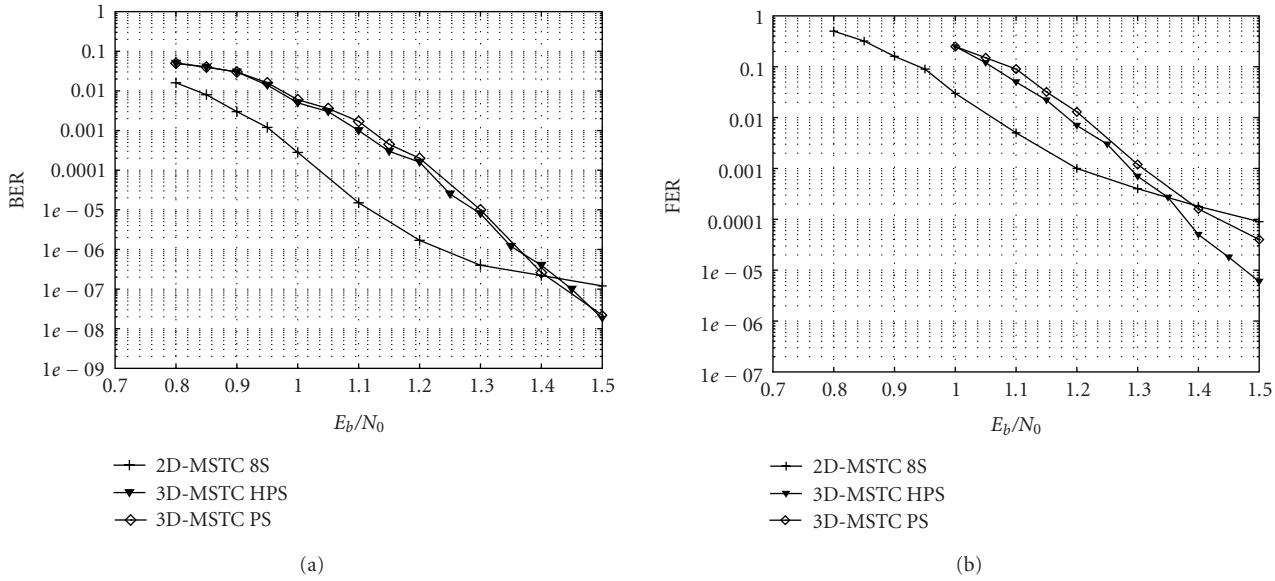


Figure 9: BER and FER comparison between the conventional PS structures and the hybrid HPS structures for (2048, 256, 8) duobinary turbo codes of rate 1/2 over an AWGN channel with the max-log-MAP algorithm (10 full iterations) and $h = 32$.

complexity compared to the 2D-MSTC. In order to adapt the 2D-MSTC decoder to the case of 3D-MSTC, with no significant additional hardware, a new serial decoding, called partial serial (PS) decoding structure, is introduced. This decoding structure is given in Figure 8, and its period is 4: during one period, dimensions 1 and 2 are decoded once, while dimension 3 is decoded twice. This structure is said to be "partial" because the third dimension only benefits from the extrinsic information of a single dimension: dimension 1 or dimension 2. Dimension 1 (or dimension 2) benefits from the extrinsic information of both dimension 2 (dimension 1, respectively) and dimension 3. These two data are added in a single memory, which will be read by the second dimension (first dimension, respectively). Hence, this structure only requires one extrinsic value per symbol to be stored. This structure reduces the memory requirements but increases the computational complexity. Compared to the ES structure, it is suboptimal, because the third dimension benefits from one dimension directly, and from the other indirectly. In terms of complexity, each decoding iteration of the partial serial (PS) method requires 4 subiterations.

Like for the HES structure, the same two sets of scaling factors are used with the PS structure leading to the hybrid partial serial (HPS) structure. As shown in Figure 9, the conclusions are the same as for the comparison between the ES and HES structures: performance improvements for the hybrid structure with less complexity.

5. PERFORMANCE

The performance of the proposed 3D-MSTC with $h = 32$ is compared to the performance of the 2D 8-state MSTC and 2D 16-state MSTC. All the codes presented in this section have a length of 4096 bits constructed with 8 slices of 256 duobinary symbols in every code dimension. They are compared through Monte Carlo simulations over an AWGN channel using a floating point max-log-MAP algorithm. Two comparisons are made at a constant decoder throughput and delay. First, the asymptotic performance of the different coding schemes is compared. For this comparison at constant throughput, the computational complexity of one decoder increases as the number of required subiterations increases. Then, in order to obtain a fair comparison, the performance is compared for the same computational complexity.

5.1. Asymptotic performance of the codes

Simulation results show that, for the MSTC codes used here, 10 iterations are sufficient to obtain most of the error-correction capabilities of the codes. In fact, additional iterations can only increase the SNR for a given BER than less than 0.1 dB. 2D-MSTCs with 8-state and 16-state trellises are compared in Figure 10 with 3D-MSTCs, for both HES and HPS decoding structures.

The FER results show that the 2D codes converge faster (0.1 dB) than the 3D code in the waterfall region. However, the union bound curves (denoted by UB) show that the error floor of the 3D code (minimum distance of 34) is slightly lower than of the 16-state code (minimum distance 32) at high SNRs. Compared to the HES decoding structure, the HPS decoding structure has a loss of less than 0.1 dB over the whole range of SNRs.

For the asymptotic performance of the codes, the computational complexity of one decoder increases as the number

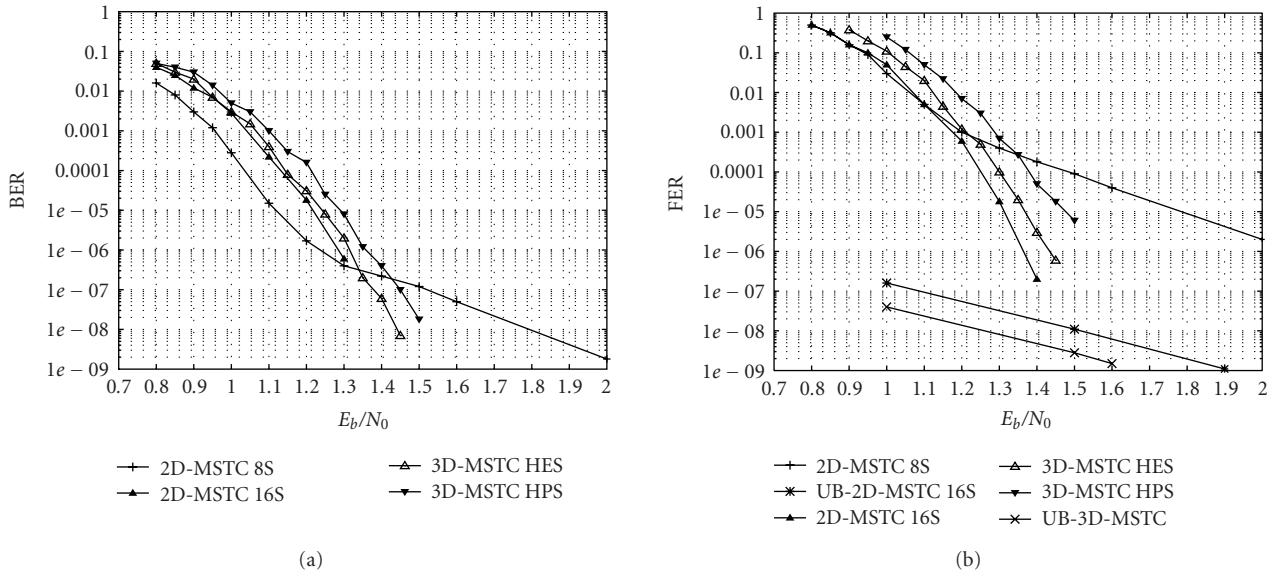


Figure 10: BER and FER of (2048, 256, 8) duobinary turbo codes of rate 1/2 over an AWGN channel with the max-log-MAP algorithm (asymptotic performance for 10 decoding iterations).

of required subiterations increases. Hence, to achieve a constant decoding throughput and delay, the corresponding decoder complexity increases as the number of required subiterations increases. This complexity comparison is analyzed in Section 6.2.

5.2. Comparison at constant computational complexity

It is obvious that the computational complexity for one decoding iteration differs between the different codes. In order to make a fair comparison, simulation results are given at constant computational complexity, that is, the same number of subiterations (decoding one dimension of the code). Thus, the decoding delay is the same for the different codes. The complexity of a 16-state trellis is assumed to be twice the complexity of an 8-state trellis. Figure 11 compares the performance for a total J of 20 subiterations of an 8-state trellis. It can been seen that, at constant complexity, HES is more efficient than the 2D 16-state MSTC over the whole range of SNRs. Moreover, HES becomes much more efficient than 2D 8-state MSTC for an FER below 10^{-4}. In addition, at a target FER of 10^{-6}, it achieves a gain of more than 0.5 dB over the 2D 8-state code. The 3D-MSTC code with HPS decoding structure shows an "error floor" at high SNRs, and therefore this decoding structure does not seem to be appropriate for this frame size and computational complexity.

When designing turbo codes, it is necessary to trade off complexity and performance. Thus, before drawing conclusions about the superiority of one over another, a comparison of the complexity of the different decoding schemes is required.

6. COMPLEXITY COMPARISON

The performance comparison of Section 5.2 pointed out that for a given computational complexity, the 3D-MSTC with HES decoding structure outperforms both 2D 8-state and 16-state codes at an FER below 10^{-4}. In terms of area, the memories of the different solutions have also to be taken into account. A generic model of area is developed in this section. This model is used to compare the different codes (of Section 5.1) with 10 full decoding iterations for their asymptotic performance.

6.1. Complexity modeling

A simple hardware complexity model is given to compare the area of the different coding schemes described in this paper. This model assumes an ASIC implementation in a 0.13 μm technology with a clock frequency F. It only takes into account the computational complexity, that is, the number P of SISOs working in parallel, and the memory area required by the turbo decoder. Moreover, there is an additional key assumption: the decoding latency of a codeword is equal to, or below, the time required to receive a new codeword (the frame duration). Thus, a parallel decoder architecture [11, 15, 30] is needed to perform the required number of iterations during the frame duration.

Since the codes are duobinary, a SISO decoder working at a frequency F can achieve a throughput of $2F$ Mbps [11]. Let J be the total number of subiterations decoded during the turbo decoding process and let D (in Mbps) be the throughput of the code. During one frame duration, every SISO decoder can decode $2 \cdot F/D$ slices. The real-time constraint implies that the SISO decoders can decode J subiterations within the reception of one frame, that is, $P \geq J \cdot D/(2 \cdot F)$.

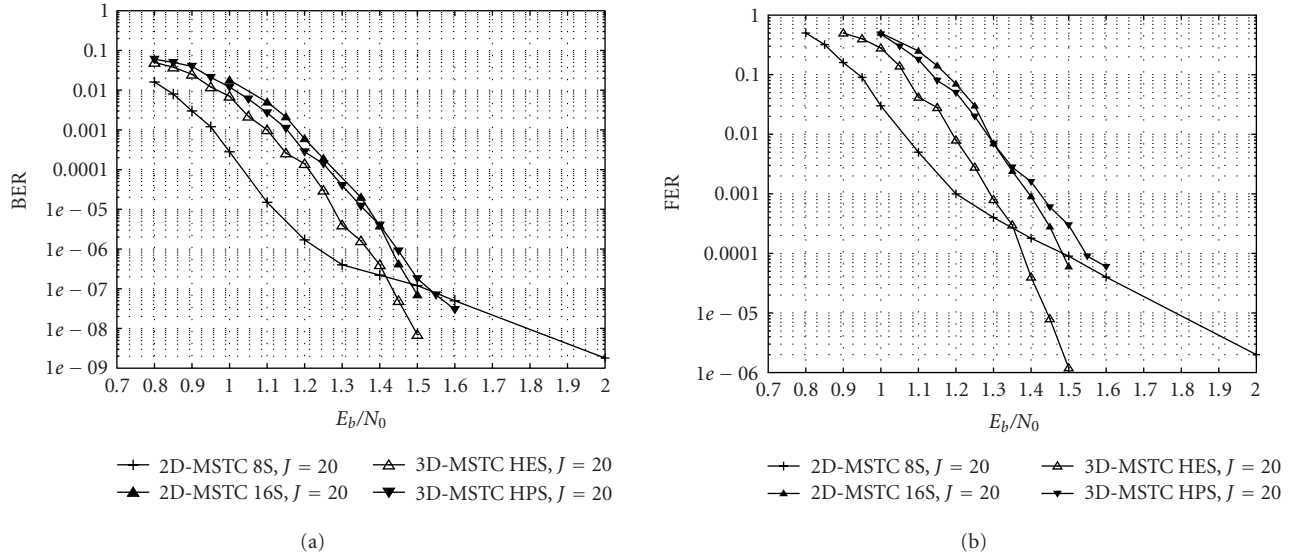


Figure 11: BER and FER of $(2048, 256, 8)$ duobinary turbo codes of rate 1/2 over an AWGN channel with the max-log-MAP algorithm (constant computational complexity $J = 20$ subiterations).

The memories required for the decoding process are composed of the intrinsic memory, which contains the output of the channel, the extrinsic memory of the current decoded frame, and an additional buffer to store the next frame while decoding the current one. The number of extrinsic memories η_E is equal to 2 for extended serial structures, 1 otherwise. The equation representing the estimation model is given in

$$A_{TD} = \left\lceil J \cdot \frac{D}{2 \cdot F} \right\rceil \cdot A_{\text{SISO}}(s) + \eta_E \text{Mem}_E(k) + 2 \cdot \text{Mem}_I(k), \tag{7}$$

where k is the number of information bits and where $\lceil x \rceil = \lfloor x \rfloor + 1$ and $\lfloor \cdot \rfloor$ denotes the integral part function. The areas $A_{\text{SISO}}(s)$ of an s-state SISO decoder are given by RTL synthesis in a 0.13 μm technology. The SISO algorithm used to compute its area is a sliding window algorithm. The areas obtained are 0.3 and $0.6mm^2$ for 8-state and 16-state SISO decoders, respectively.

6.2. Comparison

Table 2 gives the values for η_E and J for the different simulated codes presented in Section 5.1. Note that the areas of memories are also obtained by VHDL synthesis.

The values given in Table 2 show that the 3D 8-state decoders can outperform the 2D 16-state decoder in terms of complexity. Indeed, the size and the number of memories are the same for the HPS decoding structure, whereas the complexity of all SISOs is 30% higher for the 2D 16-state code. The HES decoding structure is 60% less complex in terms of SISO complexity, but the number of memories is doubled compared to the 2D 16 state code. To conclude on the relative complexity of this latter example and to choose between the HES and HPS decoding structure, we need to compare the total turbo decoder area, for a given throughput D and information frame size k. The results of this comparison are given in Table 2 for a 4096-bit information frame size and $D/2 \cdot F = 0.25$. To achieve a rate $D/2 \cdot F = 0.25$, for a 50 Mbps turbo decoder, the SISO should achieve a throughput of 200 Mbps. Since the SISO decoder is duobinary, the required clock frequency is 100 MHz, which is rather conservative. For a 100 Mbps turbo decoder, the clock frequency is doubled.

Table 2 shows that for a frame size of 4 kb, the 2D 16-state is the most complex structure. Its complexity is 75% higher than that of the 2D 8-state code. The complexities of the HES and HPS decoding structures are equivalent and 40% higher than that of the 2D 8-state code.

With increasing frame size, the size of the memory to store the extrinsic information increases. For small frame sizes up to 5000 bits, the HES decoding structure is less complex than the HPS structure. For longer block sizes, the HPS structure becomes less complex than the HES structure. The simulated performance of the HPS decoding structure shows an "error floor" for small to medium frame sizes and for a reduced number of iterations. Hence, this decoding structure may only be attractive for very long frame sizes (above 10 kb) for a number of iterations close to its asymptotic performance.

6.3. Discussion

These complexity results are preliminary results, but they show that three-dimensional turbo decoders can be efficiently implemented with considerably less complexity than the two-dimensional decoder with 16-state trellises. The architectures of the three-dimensional decoders may be improved by optimizing the tradeoff between performance and the total number of subiterations. Moreover, the extrinsic memories for the HES can be reduced by using scaling of the

Table 2: Area of the code of Section 5.1. for a ratio $D/2F = 0.25$ and $k = 4096$ bits.

Code	Decoding structure	η_E	J	SISO area	Memory area	Total area
2D 8-state	S	1	20	1.45 mm^2	0.38 + 0.24 mm^2	2.08 mm^2
2D 16-state	S	1	20	2.97 mm^2	0.38 + 0.24 mm^2	3.60 mm^2
3D 8-state	ES	2	30	2.32 mm^2	0.38 + 0.49 mm^2	3.19 mm^2
3D 8-state	PS	1	40	2.90 mm^2	0.38 + 0.24 mm^2	3.53 mm^2
3D 8-state	HES	2	25	2.03 mm^2	0.38 + 0.49 mm^2	2.90 mm^2
3D 8-state	HPS	1	30	2.32 mm^2	0.38 + 0.24 mm^2	2.95 mm^2

extrinsic information throughout the iterations as described in [31]. Another possible improvement to decrease the complexity of the 3D-MSTC is to use another constituent code for the third dimension. For example, a 4-state duobinary trellis can be used to considerably reduce the hardware complexity.

7. CONCLUSION

A new approach to designing low-complexity turbo codes for very low frame error rates and high throughput applications has been proposed. The key idea is to adapt and optimize the technique of multiple turbo codes to obtain the required error frame rate combined with a family of turbo codes, called multiple slice turbo codes (MSTC), that allow high throughput for low hardware complexity. The proposed coding scheme is based on a versatile three-dimensional multiple slice turbo code (3D-MSTC) using 8-state duobinary trellises. Simple deterministic interleavers have been used for the sake of hardware simplicity. An efficient heuristic optimization method of the interleavers has been described, leading to excellent performance. Moreover, by a novel asymmetric puncturing pattern, we have shown that convergence can be traded off against minimum distance (i.e., error floor) in order to adapt the performance of the 3D-MSTC to the requirement of the application. With the asymmetry of the puncturing pattern, it has been shown that the decoding of the third dimension (i.e., the most punctured) should be idled during the first decoding iterations. Two new adapted iterative decoding structures HES and PES have been proposed. Their performance and associated decoder complexities have been compared to 8-state and 16-state duobinary 2D-MSTC. For frame sizes up to 5000 bits, the HES decoding structure achieves lower frame error rates for less complexity than the HPS decoding structure. For a frame size of 4 kb, it has been shown that compared to a two-dimensional turbo code with 8-state trellises, the 3D-MSTC with HES decoding structure achieves a coding gain of more than 0.5 dB at a frame error rate of 10^{-6}, at the cost of a complexity increase of 50%. In addition, compared to a two-dimensional turbo code with 16-state trellises, the proposed asymmetric scheme achieves a better tradeoff performance against complexity. Future work will study the generalization of these promising results to other code rates and other frame sizes.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Proc. IEEE International Conference on Communications (ICC '93)*, vol. 2, pp. 1064–1070, Geneva, Switzerland, May 1993.

[2] DVB-RCS Standard, "Interaction channel for satellite distribution systems," *ETSI EN 301 790,* V1.2.2, pp. 21–24, December 2000.

[3] S. Lin and D. J. Costello, *Error Control Coding Fundamentals and Application*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.

[4] C. Berrou, "The ten-year-old turbo codes are entering into service," *IEEE Commun. Mag.*, vol. 41, no. 8, pp. 110–116, 2003.

[5] M. C. Valenti, "Inserting turbo code technology into the DVB satellite broadcasting system," in *Proc. 21st Century Military Communications Conference Proceedings (MILCOM '00)*, vol. 2, pp. 650–654, Los Angeles, Calif, USA, October 2000.

[6] J. D. Andersen, "Turbo codes extended with outer BCH code," *Electronics Letters*, vol. 32, no. 22, pp. 2059–2060, 1996.

[7] D. Divsalar and F. Pollara, "Multiple turbo codes for deep-space communications," TDA Progress Report 42-120, pp. 66–77, Jet Propulsion Laboratory, February 1995.

[8] C. Berrou, C. Douillard, and M. Jézéquel, "Multiple parallel concatenation of circular recursive systematic convolutional (crsc) codes," *Annals of Télécomunications*, vol. 54, no. 3-4, pp. 166–172, 1999.

[9] S. Huettinger and J. Huber, "Design of multiple-turbo-codes with transfer characteristics of component codes," in *Proc. Conference on Information Sciences and Systems (CISS '2002)*, Princeton, NJ, USA, March 2002.

[10] N. Ehtiati, M. R. Soleymani, and H. R. Sadjadpour, "Interleaver design for multiple turbo codes," in *Proc. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE '03)*, vol. 3, pp. 1605–1607, Montreal, Quebec, Canada, May 2003.

[11] D. Gnaedig, E. Boutillon, M. Jézéquel, V. C. Gaudet, and P. G. Gulak, "n multiple slice turbo codes," in *Proc. 3rd International Symposiumon Turbo Codes and Related Topics*, pp. 343–346, Brest, France, September 2003.

[12] Y. X. Cheng and Y. T. Su, "On inter-block permutation and turbo codes," in *Proc. 3rd International Symposiumon Turbo Codes and Related Topics*, pp. 107–110, Brest, France, September 2003.

[13] E. Boutillon, J. Castura, and F. R. Kschischang, "Decoder-first code design," in *Proc. 2nd International Symposium on Turbo Codes and Related Topics*, pp. 459–462, Brest, France, September 2000.

[14] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal decoding algorithm in the log domain," in *Proc. IEEE International Conference on Communications (ICC '95)*, vol. 2, pp. 1009–1013, Seattle, Wash, USA, June 1995.

[15] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel, and P. G. Gulak, "On multiple slice turbo codes," published in *Annales des Télécommunications*, January 2005.

[16] S. N. Crozier, "New high-spread high-distance interleavers for turbo-codes," in *Proc. 20th Biennial Symposium on Communications*, pp. 3–7, Kingston, Ontario, Canada, May 2000.

[17] C. Heegard and S. B. Wicker, *Turbo Coding*, Kluwer Academic Publishers, Boston, Mass, USA, 1999, pp. 50–53.

[18] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," TDA Progress Report 42-122, pp. 66–77, Jet Propulsion Laboratory, August 1995.

[19] C. Berrou, S. Vaton, M. Jézéquel, and C. Douillard, "Computing the minimum distance of linear codes by the error impulse method," in *Proc. IEEE Global Telecommunications Conference (GLOBECOM '02)*, vol. 2, pp. 1017–1020, Taipei, Taiwan, November 2002.

[20] A. S. Barbulescu and S. S. Pietrobon, "Interleaver design for turbo codes," *Electronics Letters*, vol. 30, no. 25, pp. 2107–2108, 1994.

[21] J. Hokfelt, O. Edfors, and T. Maseng, "Interleaver design for turbo codes based on the performance of iterative decoding," in *Proc. IEEE International Conference on Communications (ICC '99)*, vol. 1, pp. 93–97, Vancouver, BC, Canada, June 1999.

[22] P. C. Massey and D. J. Costello Jr., "New low-complexity turbo-like codes," in *Proc. IEEE Information Theory Workshop*, pp. 70–72, Cairns, Qld., Australia, September 2001.

[23] C. He, A. Banerjee, D. J. Costello Jr, and P. C. Massey, "On the performance of low complexity multiple turbo codes," in *Proc. 40th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, Ill, USA, October 2002.

[24] A. S. Barbulescu and S. S. Pietrobon, "Interleaver design for three dimensional turbo codes," in *Proc. IEEE International Symposium on Information Theory*, p. 37, Whistler, BC, Canada, September 1995.

[25] S. Crozier and P. Guinand, "Distance upper bounds and true minimum distance results for turbo-codes with DRP interleavers," in *Proc. 3rd International Symposiumon Turbo Codes and Related Topics*, pp. 169–172, Brest, France, September 2003.

[26] C. He, D. J. Costello Jr., A. Huebner, and K. S. Zigangirov, "Joint interleaver design for low complexity multiple turbo codes," in *41st Annual Allerton Annual Allerton Conference on Communications, Control, and Computing*, Monticello, Ill, USA, October 2003.

[27] J. Han and O. Y. Takeshita, "On the decoding structure for multiple turbo codes," in *Proc. IEEE International Symposium on Information Theory*, p. 98, Washington, DC, USA, June 2001.

[28] C. Berrou, E. Maury, and H. Gonzalez, "Which minimum hamming distance do we really need," in *Proc. 3rd International Symposiumon Turbo Codes and Related Topics*, pp. 141–148, Brest, France, September 2003.

[29] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," in *Proc. IEEE Global Telecommunications Conference (GLOBECOM '89)*, vol. 3, pp. 1680–1686, Dallas, Tex, USA, November 1989.

[30] R. Dobkin, M. Peleg, and R. Ginosar, "Parallel VLSI architecture for MAP turbo decoder," in *Proc. 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '02)*, vol. 1, pp. 384–388, Lisboa, Portugal, September 2002.

[31] R. Hoshyar, A. R. S. Bahai, and R. Tafazolli, "Finite precision turbo decoding," in *Proc. 3rd International Symposiumon Turbo Codes and Related Topics*, pp. 483–486, Brest, France, September 2003.

David Gnaedig was born in Altkirch, France, on August 28, 1978. He received the Engineering Diploma from the École Nationale Supérieure des Télécommunications (ENST), Paris, France, in 2001. Since 2002, he has been working towards his Ph.D. degree with TurboConcept, Brest, France, the Laboratoire d'Electronique et des Systèmes Temps Réels (LESTER), Lorient, France, and the École Nationale Supérieure des Télécommunications (ENST) Bretagne, Brest, France. His Ph.D thesis deals with parallel decoding techniques for high throughput turbo decoders and especially with joint code/architecture design. His research interests also include high throughput iterative decoding techniques in the field of digital communications.

Emmanuel Boutillon was born in Chatou, France, on November the 2nd, 1966. He received the Engineering Diploma from the École Nationale Supérieure des Télécommunications (ENST), Paris, France, in 1990. In 1991, he worked as an Assistant Professor in the École Multinationale Supérieure des Télécommunications in Africa (Dakar). In 1992, he joined ENST as a Research Engineer where he conducted research in the field of VLSI for digital communications. While working as an engineer, he obtained his Ph.D in 1995 from ENST. In 1998, he spent a sabbatical year at the University of Toronto, Ontario, Canada. Since 2000, he has been a Professor at the University of South Britany, Lorient, France. His current research interests are on the interactions between algorithm and architecture in the field of wireless communications. In particular, he works on turbo codes and LDPC decoders.

Michel Jézéquel was born in Saint-Renan, France, on February 26, 1960. He received the "Ingénieur" degree in electronics from the École Nationale Supérieure de l'Électronique et de ses Applications, Paris, France, in 1982. During the period 1983–1986, he was a Design Engineer at CIT ALCATEL, Lannion, France. Then, after gaining experience in a small company, he followed a one-year course about software design. In 1988, he joined the École Nationale Supérieure des Télécommunications de Bretagne where he is currently a Professor Head of the Electronics Department. His main research interest is in circuit design for digital communications. He focuses his activities on the fields of turbo encoding/decoding circuits modelling (behavioural C and synthesizable VHDL), Adaptation of the turbo principle to iterative correction of intersymbol interference, the design of interleavers, and the interaction between modulation and error correcting codes.

Appendix E

Efficient SIMD technique with parallel Max-Log-MAP Algorithm for Turbo Decoders

This appendix reproduces the paper [Gnaedig et al.-04b], which applies the MSTCs technique to the design of an efficient implementation of the Max-Log-MAP algorithm of a DSP circuit with Single Instruction Multiple Data (SIMD) capability.

Efficient SIMD technique with parallel Max-Log-MAP Algorithm for Turbo Decoders

David Gnaedig
TurboConcept / ENST Bretagne / LESTER - UBS
Technopôle Brest-Iroise
115, rue Claude Chappe
29280 Plouzané, France
david.gnaedig@turboconcept.com

Mathias Lapeyre
Université de Bretagne Sud
Centre de recherche
BP 92116
56321 Lorient Cedex, France
mathias.lapeyre1@etud.univ-ubs.fr

Florent Mouchoux
Université de Bretagne Sud
Centre de recherche
BP 92116
56321 Lorient Cedex, France
florent.mouchoux1@etud.univ-ubs.fr

Emmanuel Boutillon
Université de Bretagne Sud - Centre de recherche
BP 92116 - 56321 Lorient Cedex, France
emmanuel.boutillon@univ-ubs.fr

Abstract : This paper presents a new SIMD technique to implement efficiently on a DSP a parallel Max-Log-MAP algorithm for turbo decoders. It consists in using SIMD instructions to perform several independent trellises in parallel. This trellis parallelization is made possible by the use of an adapted two-dimensional turbo code and its parallel interleaver structure. After a brief description of the Max-Log-MAP algorithm, the implementation of a fixed point algorithm for 8-bit SIMD operations is discussed. The parallel Max-Log-MAP algorithm and the associated memory organization are described. Performance results show the effectiveness of this technique that achieves 4 times the throughput of a classical sequential implementation.

***Keywords*:** turbo codes, slice turbo code, SIMD, parallelization, Max-Log-MAP algorithm

1. INTRODUCTION

Since the introduction of turbo codes [1], there has been considerable interest in those error correcting codes. A turbo decoder consists of several concatenated soft-output decoders, each of which decodes part of the overall code and then passes "soft" reliability information in an iterative scheme. The component soft-output algorithm described in the original turbo code paper [1] is usually known as the maximum *a posteriori* (MAP), forward–backward (FB), or Bahl–Cocke–Jelinek–Raviv (BCJR) algorithm [2]. Usually, for implementation of a fast turbo-decoder on a Digital Signal Processor (DSP), its simplified version called the Max-Log-MAP algorithm working in the logarithmic domain is implemented [3].

The Max-Log-MAP algorithm is the key component of a turbo-decoder. It performs the soft decoding of a convolutional code using two finite Viterbi recursions on a trellis: the forward and the backward recursions. The basic computations involved in a trellis stage computation are Addition-Comparison-Selection (ACS) operations. The throughput of a turbo-decoder is limited by the recursions involved in the Max-Log-MAP processing of each dimension, which cannot be parallelized easily.

Loo *et al.* proposed to speed up the decoding throughput of the Max-Log-MAP algorithm on a DSP by using its Single Instruction Multiple Data (SIMD) functionality [4]. For DSP offering SIMD operations, parallel (vectorized) computations can be achieved simultaneously, on a set of data called a vector. Their SIMD technique performs, for a given trellis stage, several ACS operations. This method improves the decoding throughput but is not optimal. In fact, because of the structure of the trellis, the data need to be reordered into a single vector at the end of each trellis stage. This operation requires 2 instructions and reduces the efficiency of the SIMD technique. The use of this technique is optimal when all operations can be vectorized, *i.e.* when there is no need to pack/unpack data into a single vector to execute a SIMD operation.

To overcome this problem, this paper proposes a new technique to implement efficiently the Max-Log-MAP algorithm on a DSP. The proposed method exploits the SIMD architecture of a processor to achieve parallel processing of several Max-Log-MAP algorithms by vectorizing all operations involved in the Max-Log-MAP decoding for turbo decoders. Hence, each variable in the Max-Log-MAP algorithm is vectorized as a vector of P components: each component of the vector is associated to one of the P trellises decoded. This trellis parallelization is made possible by the use of an adapted two-dimensional turbo code proposed in [5] and called Multiple Slice Turbo Codes.

In the present paper, the adapted turbo-code is described with its interleaver properties. After a brief

description of the Max-Log-MAP algorithm, the key architectural features for implementing the algorithm with SIMD functionality are given (memory organization and SIMD operations).

1. MULTIPLE SLICE TURBO CODES

The idea of Multiple Slice Turbo Codes (MSTC) was proposed by Gnaedig et al. [5]. The motivation of MSTCs construction was to design an adapted turbo code suitable for an efficient parallel implementation. It is made possible by an increase by a factor *P* (the number of slices) of the decoding parallelism of the turbo-decoder without memory duplication. This technique leads to efficient parallel turbo decoder FPGA or ASIC implementations.

An adapted turbo code can also be designed to decode efficiently a turbo decoder on a DSP by using SIMD techniques. It has been noticed that the properties of Multiple Slice Turbo Codes are suitable for efficient vector operations on a DSP. Indeed, this adapted turbo-code is constructed, in each dimension, as the parallel concatenation of *P* independent Circular Recursive Systematic Convolutional (CRSC) codes, called slices. These *P* slices can be decoded in parallel with a Max-Log-MAP using vectorized operations.

Moreover, an appropriate parallel interleaver is also proposed in [5]. This parallel property avoids conflicts in parallel memory accesses. Therefore, no memory duplication is required and a single memory can be used. In addition, thanks to its structure, the interleaver maintains the vector organization between the natural and the interleaved order. This vectorization property leads in an efficient implementation, because there is no need to shuffle data by unpacking / repacking into vectors between half-iterations.

In this section, the construction of Multiple Slice Turbo Codes is described. Then, it will be shown how the structure and the design of the parallel interleaver enables a vectorization and a parallellization of the decoding.

1.1. Code construction

Multiple Slice Turbo Codes are constructed as follows. An information frame of *N m*-binary symbols is divided into *P* blocks (called "slices") of *M* symbols, where $N = M \cdot P$. The resulting turbo code is denoted (*N*,*M*,*P*). As with a classical convolutional turbo code, the coding process is first performed in the natural order to produce the coded symbols of the first dimension. Each slice is encoded independently with a Circular Recursive Systematic Convolutional (CRSC) code. The information frame is then permuted by an *N* symbol interleaver. The permuted frame is again divided into *P* slices of size *M* and each of them is encoded independently with a CRSC code to produce the coded symbols of the second dimension. Puncturing is applied to generate the desired code rate.

The interleaver is constructed jointly with the memory organization to allow parallel decoding of the *P* slices. In other words, at each symbol cycle *k*, the interleaver structure allows the *P* decoders to read and write the *P* necessary data symbols from the *P* Memory Banks MB_0, MB_1,…, MB_{P-1} without conflict. Indeed, only one read can be made at any given time from a single port memory: in order to access *P* data symbols in parallel, *P* memory banks are necessary. With the solution described in the present paper, the degree of parallelism can be chosen according to the requirements of the application.

The next section presents the parallel interleaver construction, ensuring the parallelism constraint while maintaining good performance.

1.2. Parallel interleaver design

The interleaver structure is mapped onto a hard-ware architecture allowing a parallel decoding process. Figure 1 presents the interleaver structure used to construct a multiple slice turbo code constrained by decoding parallelism.

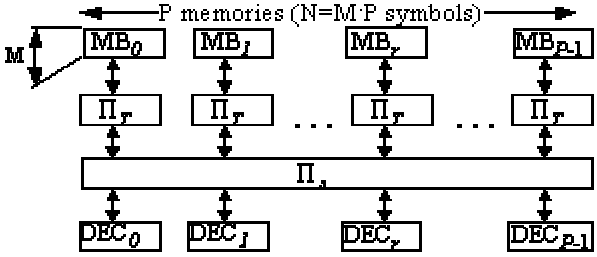


Figure 1: Interleaver structure for the (*N*,*M*,*P*) code

Let *l* and *k* denote the indices of the symbols in the natural and interleaved order, respectively. The coding process is performed in the natural order on independent consecutive blocks of *M* symbols. The symbol with index *l* is used in slice $\lfloor l/M \rfloor$ at temporal index time *l* mod *M,* where $\lfloor . \rfloor$ denotes the integral part function. Likewise, in the interleaved order, the symbol with index *k* is used in slice $r = \lfloor k/M \rfloor$ at temporal index $t = k \bmod M$. Note that $k = M \cdot r + t$, where $r \in \{0..P-1\}$ and $t \in \{0..M-1\}$. For each symbol with index *k* in the interleaved order, the permutation Π associates a corresponding symbol in the natural order with index $l = \Pi(k) = \Pi(t,r)$. The interleaver function can be split into two levels: a spatial permutation $\Pi_S(t,r)$ (ranging from 0 to *P*-1) and a temporal permutation $\Pi_T(t,r)$ (ranging from 0 to *M*-1), as defined in (1) and described in Figure 1.

$$\Pi(k) = \Pi(t,r) = \Pi_S(t,r) \cdot M + \Pi_T(t,r) \qquad (1)$$

The symbol at index *k* in the interleaved order is read from the memory bank $\Pi_S(t,r)$ at address $\Pi_T(t,r)$. While decoding the first dimension of the code, the frame is processed in the natural order . The spatial and temporal permutations are then simply replaced by *Identity* functions. The spatial permutation allows the *P* data read out to be transferred to the *P* SISO units

(denoted SISO in Figure 1). Decoder *r* receives the data from memory bank $\Pi_S(t,r)$ at instant *t*. This spatial permutation guarantees the parallel property of the interleaver.

1.3. Design of the permutations

The temporal $\Pi_T(t,r)$ and $\Pi_S(t,r)$ spatial permutations are chosen to simplify the implementation on a DSP (address computation, memory organization, simple DSP operations, ...), while maintaining good performance. To guarantee the vectorization property of the interleaver, the same temporal permutation is chosen for all memory banks. Hence, $\Pi_T(t,r) = \Pi_T(t)$ depends only on the temporal index *t*. The temporal permutation is then given by:

$$\Pi_T(t) = \alpha \cdot t + \beta(t \bmod 4) \bmod M \,, \qquad (2)$$

where α is relatively prime with *M*, and $(\beta(i))_{0\le i<4}$ are four coefficients inferior or equal to *M,* which verify that their values modulo 4 are all different.

The spatial permutation is defined as a circular rotation:

$$\Pi_S(t,r) = \left(A(t \bmod P) + r\right) \bmod P \,, \qquad (3)$$

where *A* is a bijection of variable $t \in \{0,...,P-1\}$ to $\{0,...,P-1\}$. The choice of the temporal and spatial permutation (2) and (3) can be easily implemented by additions and circular shift operations, respectively. Their optimization described in [6] in more details leads to very good performance, where it has also been shown that this adapted turbo code introduces no performance degradation compared to a conventional turbo code.

1.4. A simple example of an interleaver

Let us construct a simple (18,6,3) code to clarify the interleaver construction. Let the temporal permutation be $\Pi_T(t) = \{1,4,3,2,5,0\}$ and the spatial permutation a circular shift of amplitude $A(t \bmod 3)$, i.e. the slice of index *r* is associated to the memory bank of index $\Pi_S(t,r) = \left(A(t \bmod 3) + r\right) \bmod 3$, with $A(t \bmod 3) = \{2,0,1\}$. The spatial permutation is then bijective and 3-periodic.

The interleaver is illustrated on Figure 2, which shows the permutations for the 3 temporal indexes *t*=0 (2.a), *t*=1 (2.b) and *t*=5 (2.c). The 18 symbols in the natural order are separated into 3 slices of 6 symbols corresponding to the first dimension. In the second dimension, at temporal index *t*, symbols $\Pi_T(t)$ are selected from the 3 slices of the first dimension, and then permuted by the spatial permutation $\Pi_S(t,r)$. For example, at temporal index $t = 0$ (a), symbols at index $\Pi_T(0) = 1$ are selected. Then, they are shifted with an amplitude $A(0 \bmod 3) = 2$. Thus, symbols 1 from slices 0,1 and 2 of the first dimension go to slices 2, 0 and 1 of the second dimension respectively.

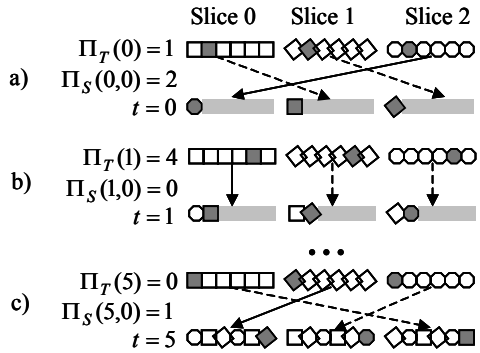


Figure 2 : A basic example of a (18,6,3) code.

2. THE MAX-LOG-MAP ALGORITHM

In this section the Max-Log-MAP algorithm for a binary MSTC used for decoding each slice of *M* bits is described by using the same notations than in [11].

The CRSC code is represented by a circular trellis of *M* stages. For each information bit u_k, the convolutional encoder produces the coded bits $X_k = (x_k^i)_{0\le i<r}$, where *r* is the number of bits produced by the convolutional code. These coded bits are transmitted over an additive white gaussian noise channel (AWGN). The decoder receives the noisy bits $Y_k = (y_k^i)_{0\le i<r}$ corresponding to the transmitted bits.

The Max-Log-MAP algorithm [3] provides for each information bit u_k, $k = 0,..., M-1$ a soft output $L(u_k)$, which represents the *a posteriori* probability for the bit u_k. For iterative decoding, another soft output $L_e(u_k)$ called the extrinsic information is computed from $L(u_k)$. This latter is computed by using the all received values of the channel $Y_k = (y_k^i)_{0\le i<r}$, $k = 0,..., M$-1 and the *a priori* information $L_a(u_k)$ provided by the other decoder, which is its extrinsic soft output.

The output of the channel and the *a priori* decoder inputs are used to compute the branch metrics (are used to process the forward and backward recursions.) between state *s*' at stage *k* and *s* at stage *k*+1 of the trellis by a scalar product between the received vector Y_k at time *k*:

$$c_k(s',s) = \sum_{i=1}^{r} x_k^i(s',s) \cdot y_k^i + L_a(u_k) \qquad (4)$$

The soft estimate $L(u_k)$ is computed by exhaustively exploring all possible paths in the trellis using a forward recursion and a backward recursion. This algorithm consists of three steps (Figure 3):

- Forward recursion: The forward state metrics a_k are recursively calculated using the bits in an increasing order from 0 to *M*-1:

$$a_k(s) = \max_{(s,s')} \left(a_{k-1}(s') + c_k(s',s)\right), \qquad (5)$$

where s' and s are the generic states at the (k-1)th and kth nodes respectively.
The forward state metrics are stored in an internal memory in order to be used to compute the soft output.

- Backward recursion: The backward state metrics b_k are recursively computed using the bits in a decreasing order from *M-1* to *0*.

$$b_k(s) = \max_{(s,s')}\left(b_{k+1}(s') + c_{k+1}(s,s')\right) \quad (6)$$

- Soft-Output Computation. The soft output for each bit at time k is computed by using the backward state metric b_k and the corresponding forward state metric a_{k-1} read from the memory.

$$L(u_k) = \max_{\substack{(s',s)\\ u_k=1}}\left(a_{k-1}(s') + c_k(s',s) + b_k(s)\right) - \max_{\substack{(s',s)\\ u_k=0}}\left(a_{k-1}(s') + c_k(s',s) + b_k(s)\right) \quad (7)$$

The extrinsic soft output is computed from the soft ouput $L(u_k)$ by subtracting the input of the decoder:

$$L_e(u_k) = L(u_k) - L_a(u_k) - 2 \cdot y_k^1, \quad (8)$$

where y_k^1 is the received bit corresponding to the transmitted bit $x_k^1 = u_k$.

Since the trellis of the code is circular, the initial state metrics of the forward and backward recursions a_0 and b_{M-1} are provided by the final state metrics of the previous iterations (see Figure 3). At the first iteration, these initial state metrics are set to the all zero vector.

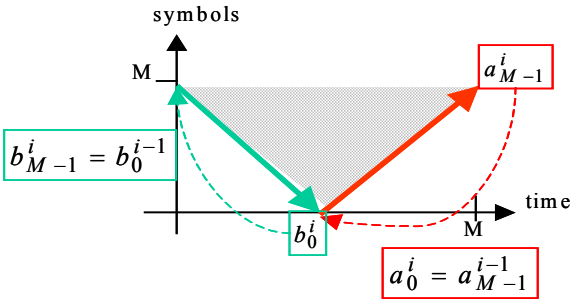


Figure 3: Graphical representation of the Forward-Backward Max-Log-MAP algorithm

3. TURBO DECODER IMPLEMENTATION

In this section, the algorithm described in the previous section is processed in parallel on several trellises by using the SIMD technique of the DSP. After a brief description of DSP architectures and SIMD techniques, the implementation of the turbo decoder is described.

3.1. VLIW and SIMD techniques for DSP

Current DSPs can achieve several thousands of million of operations per seconds. This high number of operations is achieved thanks to two architectural concepts. The first concept consists in Very Long Instruction Word (VLIW). It consists in processing several operations simultaneously by several processing units. Usually, high performance fixed point DSPs are processing operations on 32 bits. The second architectural concept to increase the parallelism in the DSP is to use an Single Instruction Multiple Data (SIMD) technique. This technique enables to process two 16-bit or four 8-bit identical operations with a single instruction on a single 32-bit processing unit. The set of two or four data that are processed simultaneously is called a vector. The operations processing a vector of data is said to be "vectorized".

It is not an easy task to achieve an efficient utilization of the VLIW technique, *i.e.* to write a code that uses all processing units continuously. Hence, this task is left to the compiler. The turbo decoder is thus described with a fixed point C code. This C code uses explicitly vector operations and has an adapted memory organization suitable for efficient use of the SIMD capability.

3.2. Fixed point Max-Log-MAP algorithm

When using vectorized operations, the number of bits used to code internal variable has to be lower or equal to the number of bit of a SIMD operation. For example, for a 32-bit processor processing 4 operations of 8-bit on one SIMD processing unit, the maximum number of bits for the internal variables is 8. If one internal variable is higher than 8, the corresponding operations can be done in parallel only by two 16-bit operations and additional cycles are needed to unpack and repack the data after and before this operation. This solution decreases the efficiency of the parallel implementation. The other solution consists in using 16-bits operations for the whole algorithm, but the parellelism and hence the throughput are divided by two. This motivates the need for techniques that reduce to 8 bits the internal precision of the Max-Log-MAP algorithm.

The conversion of a floating point Max-Log-MAP algorithm to a fixed point algorithm has been widely described in the literature [8]. The key parameter is the precision of the inputs of the decoder, *i.e.*. the number of bit wd used to code the decoder inputs. This parameter incluences the performances and also determines the internal size w_n of the state metrics.

The problem that arises when implementing a Viterbi recursion, is the increase of the state metrics. This problem has been intensively studied for the Viterbi algorithm [9] and solutions using rescaling or modulo 2^{wn} arithmetic are widely used [10]. These techniques are based on the fact that, at every instant, the dynamic range of a state metric (i.e., the difference between the state metrics with the highest and lowest values), is bounded. This upper bound is derived from the constraint length of the code and the maximum value of

the branch metrics. It enables us to use a simple rescaling technique. For our fixed point implementation, the max operations in equations (5-7) are replaced by min operations, by changing the sign of the branch metrics (4). Thus, at each trellis state, the state metrics are normalized by subtracting the minimal state metric. This solution leads to the least number of bits w_n to store the state metrics.

For our application, the constituent code used is a duo-binary code of rate 2/3 [7]. The equations of the Max-Log-MAP algorithm for a duo-binary turbo code are similar to (4)-(7). It produces 4 reliabilities for the 4 possible transmitted symbols and the number of branches leaving each trellis state is equal to 4.

After demodulation, the ouput of the channel are provided to the decoder as quantized unsigned values of w_d bits. The extrinsic information are also coded on $w_e=w_d$ bits. Hence, from (4), the branch metric can be coded on w_d+2 bits. In addition, the dynamic range of a state metrics is bounded by twice the maximum value of the branch metrics. Thus, the state metrics are coded on w_d+3 bits for equations for (5) and (6). In equation (7), the addition of the forward state metric and the backward state metrics requires w_d+4 bits to code the soft output. After computation of the extrinsic information using (8), the 4 values of extrinsic information for each symbol are normalized by subtracting the minimum value. Then these unsigned values are thresholded to be stored on $w_e=w_d$ bits. From this study, the maximum number of bits for an internal value of the decoder is w_d+4. This number must be inferior or equal to the maximum number of bits for a single SIMD operation, *i.e.* 8. This leads to $w_d = 4$ bits to code the inputs of the decoder. In section 4, the performance of the decoder with $w_d = 4$ bits is compared to those of an infinite precision decoder.

3.3. Parallel Max-Log-MAP algorithm and memory organization

The fixed point Max-Log-MAP algorithm described in the previous section is implemented with the SIMD technique. For a DSP with a SIMD parallelism enabling *P*-component vector operations, the adapted turbo code is designed with a parallelism of *P*. Hence, *P* different trellises corresponding to *P* slices can be decoded in parallel. Each component of an internal vector corresponds to an internal variable of the Max-Log-MAP algorithm for one trellis. Hence, all operations of the Max-Log-MAP algorithm are vectorized. Therefore, a memory organization is needed for the input and outputs of the Max-Log-MAP algorithm. This memory organization allows us to simply use vectorized operations for the parallel Max-Log-MAP algorithms, without extra cycles for packing / unpacking of data. In addition, this memory organization verifies the vectorization property of the interleaver. The *P* inputs of the Max-Log-MAP algorithm are regrouped in vectors of *P* component, with one vector for *P* data that are used at the same time, in the same vector operation.

Because of the vectorization property of the interleaver, only one address computation and one memory access in needed to read one vector from the memory. Only a circular shift is needed in the interleaved order to implement the spatial rotation of the interleaver. This operation can be done in a single cycle.

4. RESULTS

The trellis parallelization technique has been implemented on a DSP TMS320C6416: four parallel Max-Log-MAP algorithms are performed on $P = 4$ independent trellises by using 8-bit SIMD operations.

4.1. Implementation

The fixed point Max-Log-MAP algorithm with $w_d = 4$ bits is implemented in C language. The SIMD instructions are specified by using intrinsic functions. Program 1 gives the corresponding C code implementing the forward recursion. To reduce the complexity, the rescaling of the state metrics is not performed at each trellis stage. They are only rescaled when at least one of the state metrics exceed 2^{w_d-1}.

The SIMD implementation of the backward recursion and the extrinsic information computation are similar to the one of the forward recursion. They are therefore not described here.

```
test = 0x00000000;
/* ACS */
for(s = 0; s < number_state; s++) {
   /* Addition */
   for (b = 0; b < 4; b++) {
      met_br = _add4(br_met[s][b],ap_in[b]);
      fwd_br[s][i] = _add4(fwd_in[s],met_br);
   }
   /* Comparison and Selection */
   tmp0 = _minu4(
      fwd_br[previous_state[s][0]][0],
      fwd_br[previous_state[s][1]][1]);
   tmp1 = _minu4(
      fwd_br[previous_state[s][2]][2],
      fwd_br[previous_state[s][3]][3]);
   fwd_out[s] = _minu4(tmp0,tmp1);
   /* Test */
   if (!test)
      test |= fwd_out[s] & 0x80808080;
}

/* Rescaling */
if (test) {
  fwd_min = fwd_out[0];
   for(s = 1; s < number_state; s++) {
      fwd_min = _min4(fwd_min, fwd_out[s]);
   }
   for(s = 0; s < number_state; s++) {
      fwd_out[s] = _sub4(fwd_out[s], fwd_min);
   }
}
```

Program 1: SIMD description of the forward algorithm.

4.2. Performance

The performance of the fixed point algorithm with $wd = 4$ is compared to the performance of an infinite precision floating point algorithm. The simulated performance of turbo decoders using these algorithms are shown in Figure 4 for the same duo-binary turbo code of 512 bits constructed with $P = 4$ slices of 64 symbols. Eight iterations have been performed for the two algorithms.

The performance degradation of the algorithm with $wd = 4$ bits is only 0.1 dB compared to the infinite precision algorithm, which is the same than for a fixed point algorithm with $wd = 8$. This result shows that increasing the number of input bits of the algorithm does not provide a huge improvement. It has been shown in section 0 that with $wd > 4$ bits, only 16-bit SIMD operations can be used. This reduces the thoughput of the Max-Log-MAP algorithm by a factor of two.

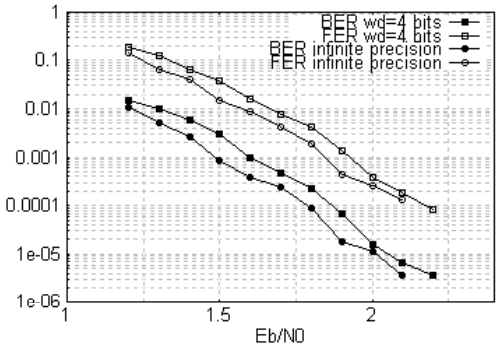


Figure 4: Performance comparison of the Max-Log-MAP algorithm for a (256,64,4) duo-binary turbo-code: the algorithm with wd = 4 inputs bits is compared to the infinite precision algorithm.

4.3. Throughput

The Max-Log-MAP algorithm has been simulated on the DSP for the duo-binary turbo code (256,64,4). The number of cycles for performing the Max-Log-MAP algorithm for one dimension and 8 iterations respectively are given in Table 1. The number of cycles does not increase linearly with the number of iterations, because the branch metrics are computed only once at the first iteration and stored into the memory.

	Number of cycles	Throughput @ 600 MHz
One dimension	292 554	1050 kbits/s
Turbo decoder @ 8 it.	1 927 344	160 kbits/s

Table 1: Number of cycles and throughput for the Parallel Max-Log-MAP algorithm and the turbo-decoder with 8 decoding iterations.

The implementation results shows that for a 600 MHz clock frequency, the parallel Max-Log-MAP algorithm used in a turbo decoder with 8 decoding iterations achieves a throughput of 160 kbits/s. It is exactly 4 times the throughput of a sequential implementation without using the SIMD technique.

5. CONCLUSION

A new SIMD technique for the Max-Log-MAP algorithm has been presented in this paper. This technique uses an adapted turbo, whose interleaver structure enables us to use SIMD operations for all computations and memory accesses. The degradation in performance with 8-bit SIMD operations is only 0.1 dB from an infinite precision algorithm with a throughput of 160 kbits/s, 4 times that of a sequential implementation. With 16-bit SIMD operations there is no performance degradation, but the throughput is only doubled. Futher work includes designing a complete turbo decoder and optimizing its implementation to further increase the thoughput.

6. REFERENCES

[1] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes", *Proc. ICC'93*, Geneva, Switzerland, pp. 1064-1070, May 1993.

[2] L.R Bahl, J. Cocke, F. Jelinek, J. Raviv, " Optimal decoding of linear codes for minimizing symbol error rate ", IEEE Transactions on Information Theory, pp. 284-287, mars 1974.

[3] L. Papke and P. Robertson, "A Viterbi algorithm with soft-decision outputs and its applications", *Proc. IEEE Int. Conf. Commun.*, Dallas, pp. 102-106, June 1996.

[4] K.K. Loo, T. Alukaidey, S.A. Jimaa, K. Salman, *"SIMD Technique for Implementing the Max-Log-MAP Algorithm",* GSPx 2003.

[5] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel, P. G. Gulak, "On Multiple Slice Turbo Codes", in *Proc. International Symposium on Turbo Codes and Related Topics*, Brest, pp. 343-346, Sept. 2003.

[6] ____ , "On Multiple Slice Turbo Codes", *submitted to Annales des Télécommunications*

[7] DVB-RCS Standard, "Interaction channel for satellite distribution systems", *ETSI EN 301 790*, V1.2.2, pp. 21-24, Dec 2000.

[8] G. Montorsi and S. Benedetto, "Design of fixed-point iterative decoders for concatenated codes with interleavers," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 871–882, May 2001.

[9] C. B. Shung, P. H. Siegel, G. Ungerboeck, and H. K. Thapar, "VLSI architectures for metric normalization in the Viterbi algorithm," in Proc. *IEEE Int. Conf. Communications* (ICC '90), vol. 4, Atlanta, GA, Apr. 16–19, 1990, pp. 1723–1728.

[10] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo codes," IEEE Trans. VLSI Syst., vol. 7, pp. 369–379, Sept. 1999.

[11] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 260–264, Feb. 1998.

Appendix F

Résumé étendu

Sommaire :

Appendix F. Résumé étendu

Introduction

Les turbocodes sont des codes obtenus par une concaténation de plusieurs codes convolutifs séparés par des entrelaceurs. En 1993, ils ont révolutionné le domaine du codage correcteur d'erreurs en s'approchant à quelques dixièmes de décibels de la limite théorique de Shannon. Ces performances sont d'autant plus remarquables que le principe itératif permet d'en effectuer le décodage avec une complexité matérielle limitée. Le succès des turbocodes s'est traduit par leur introduction dans plusieurs standards de communication. Dès lors, les besoins croissants dans le domaine des réseaux large bande, nécessitent des implantations hauts débits qui posent de nouvelles problématiques.

L'objectif de cette thèse est d'étudier des architectures de décodage à haut débit de concaténations parallèles de codes convolutifs offrant le meilleur compromis entre débit et complexité. Le problème d'implantation du décodeur étant intimement lié à la construction du turbo code et particulièrement de l'entrelaceur, nous avons abordé ces deux aspects conjointement. Nous avons exploré un large spectre de possibilités de l'espace de conception allant de la construction conjointe du code et du décodeur à l'optimisation directe des architectures de décodage pour un code ou un ensemble de codes prédéfinis.

Afin d’aborder ce problème de conception d’architectures de décodage efficaces, nous proposons dans le deuxième chapitre un modèle simple permettant d'exprimer le débit et l'efficacité d'une architecture. Ce modèle appliqué au turbo-décodage met en évidence trois paramètres caractéristiques ayant un impact sur le débit et l'efficacité du décodeur : le degré de parallélisme, le taux d'utilisation (activité) des unités de calcul et la fréquence d'horloge. Les contributions importantes de cette thèse résultent des recherches effectuées dans les trois directions indiquées par ces paramètres caractéristiques. Notre première contribution concerne l’augmentation des ressources de calculs par l’introduction de parallélisme dans le décodage : plusieurs processeurs travaillent en parallèle sur la même trame codée. Afin de résoudre les problèmes d’accès concurrents aux mémoires qui en résultent, nous avons proposé une nouvelle famille de codes appelés turbocodes à roulettes. Ils permettent dès l’étape de la conception du code de résoudre les conflits pré-cités par la prise en compte de l’architecture de décodage associée. Notre deuxième contribution concerne l’augmentation de l’activité des processeurs de calculs, qui se dégrade fortement dans le contexte des architectures parallèles. Cette problématique est nouvelle et n’a pas été traitée dans la littérature auparavant. Nous avons proposé plusieurs solutions originales permettant d’augmenter l’activité des processeurs. Enfin, la troisième contribution de cette thèse est la proposition d’une méthode originale permettant d'augmenter la fréquence d'horloge du décodeur. L’ensemble des

contributions de cette thèse est appliqué à la réalisation d'un décodeur à haut débit sur un circuit FPGA décrit dans le dernier chapitre.

F.1 Contexte et état de l'art

Ce premier chapitre introduit les principaux concepts des turbocodes et de leur décodage et introduit les notations et algorithmes utilisés dans cette thèse. Après une brève description des fondamentaux sur le codage de canal, nous introduisons les codes convolutifs et en particulier la classe des codes systématiques récursifs, utilisés dans les constructions de turbocodes. Nous exposons également les codes circulaires qui permettent de terminer le treillis de manière efficace. Puis, nous décrivons le concept de base des turbocodes construits comme une concaténation parallèle de deux codes convolutifs identiques séparés par un entrelaceur. Nous soulignons en particulier le caractère fondamental de ce dernier et de ses caractéristiques permettant d'assurer de bonnes performances au code. Cette étude nous conduit à analyser les performances de quelques catégories d'entrelaceurs. La puissance des turbocodes est en grande partie due au principe de décodage itératif faisant intervenir un échange d'informations extrinsèques entre plusieurs décodeurs à entrées et sorties pondérées (SISO). Ce concept d'information extrinsèque produit par le décodeur SISO est fondamental, car son utilisation permet au décodage itératif de converger après quelques itérations vers une decision commune, qui correspond au mot de code émis si toutes les erreurs on été corrigées. Cette information extrinsèque peut être calculée par différents algorithmes que nous décrivons opérant sur un treillis représentatif du code. Nous détaillons plus particulièrement l'algorithme optimal symbole par symbole appelé algorithme BCJR [Bahl *et al.*-74]. Nous dérivons son expression logarithmique simplifiée appelée algorithme Max-Log-MAP, qui fait intervenir deux récursions de Viterbi (Aller et Retour) qui sont combinées pour produire les informations extrinsèques. Enfin, nous en étudions la complexité et le séquencement.

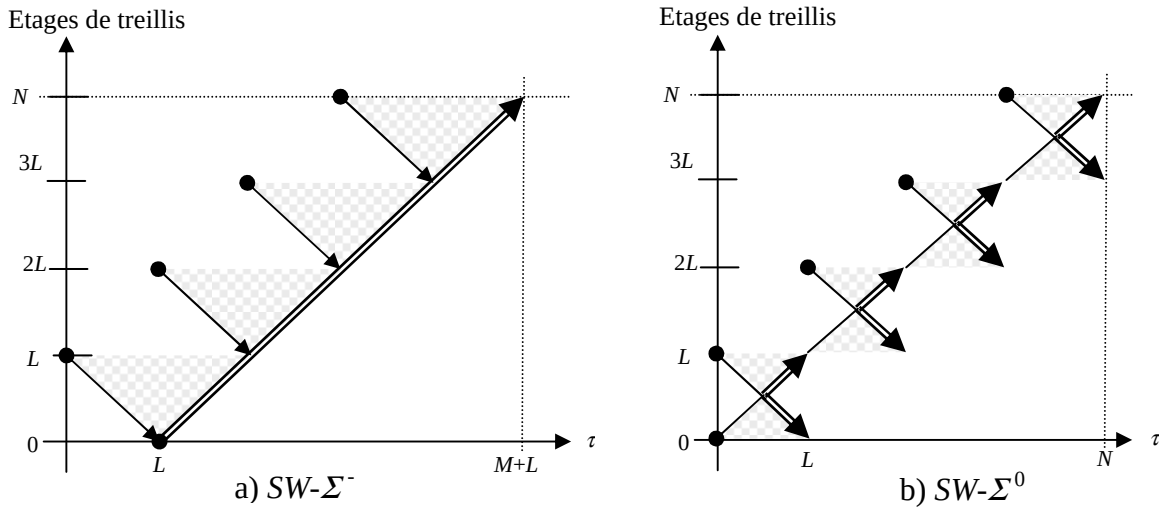


Figure F-7 : Séquencement à fenêtre glissante pour une demi-itération : a) *SW-Σ⁻*; b) *SW-Σ⁰*

Enfin, après un bref rappel historique, nous simplifions cet algorithme à sa plus simple expression: un algorithme à fenêtre glissante dont deux séquencements sont décrits sur la Figure F-7 : *SW-Σ⁻* ou *SW-Σ⁰*. Bien que ces deux séquencements aient une complexité

algorithmique identique, leur temps de décodage diffèrent ce qui induit des architectures matérielles différentes.

F.2 Architectures matérielles pour décodage à haut débit

Dans ce chapitre nous introduisons la notion d'architecture permettant d'implanter matériellement un algorithme. Nous proposons un modèle général d'architecture permettant d'en exprimer le débit et l'efficacité, définie comme le débit divisé par la complexité. Nous montrons que le débit peut s'exprimer en fonction de grandeurs caractéristiques de l'algorithme et de l'architecture : f_{clk} la fréquence d'horloge, n_a la complexité algorithmique de l'algorithme, Γ le nombre de ressources de calculs et α l'activité des ressources de calcul, c'est-à-dire le taux d'utilisation desites ressources. Après un bref état de l'art des implantations de turbo-décodeurs, nous évaluons un modèle de complexité algorithmique de l'algorithme Max-Log-MAP, dont nous validons la pertinence par comparaison avec des résultats de synthèse sur différentes cibles ASICs et FPGAs. L'application du modèle d'architecture au turbo-décodage de code *m*-binaire se traduit par l'expression suivante du débit :

$$D_b = f_{clk} \cdot \alpha(\Gamma) \cdot m \cdot \frac{\Gamma}{2 \cdot I_{\max} \cdot \chi_u}. \qquad \textbf{(F-1)}$$

Cette relation fait apparaître les paramètres fondamentaux de l'architecture sur lesquels nous pouvons agir indépendamment pour augmenter le débit :

- choisir un algorithme de décodage avec une faible complexité algorithmique χ_u;
- réduire le nombre d'itérations soit de manière statique (limiter le nombre maximal d'itérations I_{max}) ou dynamique par l'utilisation d'un critère d'arrêt;
- augmenter la fréquence d'horloge du circuit, qui est limitée par le calcul récursif des métriques de nœuds comprenant des opérations d'Addition-Comparaison-Sélection (ACS);
- augmenter les ressources de calcul Γ;
- augmenter l'activité α des processeurs.

Les trois derniers points mettent en évidence trois paramètres importants qui ouvrent autant de perspectives de recherche que nous abordons dans les chapitres qui suivent. Concernant l'augmentation des ressources de calcul, nous décrivons dans un premier temps la première solution qui consistait à dupliquer en cascade le décodeur élémentaire autant de fois que le nombre d'itérations. Cette architecture série permet une augmentation linéaire du débit avec la complexité mais elle est fortement pénalisée par la duplication des mémoires d'observation et extrinsèques. Afin d'éviter la duplication des mémoires, l'architecture parallèle a été introduite. Elle consiste à diviser le treillis de taille N en P étages de taille M ($N = M \cdot P$) et de décoder chacune des P sections de treillis de manière indépendante par un processeur dédié. Nous décrivons une architecture des mémoires permettant des accès concurrents à la mémoire des informations extrinsèques par les P processeurs, et nous montrons comment la présence

de l'entrelaceur conduit nécessairement à des conflits d'accès aux mémoires ce qui représente le problème majeur à résoudre pour les architectures parallèles. Puis, nous quantifions la décroissance de l'activité d'une architecture parallèle lorsque le nombre de processeurs augmente ou lorsque la taille de trame diminue. Cette forte décroissance s'observe pour le séquencement $SW\text{-}\Sigma^{-}$ bien que ce dernier puisse atteindre l'activité maximale de 1. Le séquencement $SW\text{-}\Sigma^{0}$ est moins sensible à ces paramètres, mais son activité est bornée à cause d'une sous-utilisation des modules de calcul d'informations extrinsèques.

F.3 Décodage parallèle de turbocodes

Dans ce chapitre nous traitons la résolution des conflits de parallélisme résultant des accès concurrents aux mémoires par les processeurs d'une architecture parallèle. Dans un premier temps, nous proposons une classification des différentes solutions proposées en trois catégories selon le moment de la conception auquel ils résolvent les conflits: à l'exécution, à la compilation (conception de l'architecture) où à la conception (du code). Notre contribution se situe dans la troisième catégorie de solutions par la proposition d'un nouveau schéma de codage appelé *turbocodes à roulette*. Il s'agit d'un turbocode convolutif conventionnel associé

- à un entrelaceur hiérarchique à deux niveaux dont la structure est adaptée à l'architecture de décodage parallèle choisie et auquel on applique des contraintes permettant de résoudre tous les conflits de parallélisme;
- à une modification du code constituant permettant d'exhiber plus de parallélisme : l'utilisation de codes circulaires indépendants appelés roulettes facilite la conception de l'entrelaceur hiérarchique.

Nous détaillons ensuite la conception des turbocodes à roulettes et en particulier l'optimisation de l'entrelaceur permettant de garantir de bonnes performances au code. Ce travail nous conduit à des équations d'entrelacement simples à mettre en œuvre dans un décodeur. Nous terminons ce chapitre par des courbes de performances qui montrent l'absence de dégradation de performance due à l'introduction de contraintes dans l'entrelaceur pour satisfaire une architecture de décodage parallèle. Après une comparaison avec d'autres entrelaceurs structurés permettant un décodage parallèle et proposés dans la littérature, nous concluons en comparant l'ensemble de toutes les solutions décrites dans ce chapitre (execution, compilation, conception) et en donnant leurs intérêts et domaines d'application respectifs. Les résultats de ce chapitre ont été publiés en partie dans [Gnaedig *et al*-03], plus en détails dans [Gnaedig *et al*-05a] et [Gnaedig *et al*-05b]. Enfin, les turbocodes à roulettes ont fait l'objet d'un dépôt de brevet [Boutillon *et al*-02].

F.4 Augmentation de l'activité des architectures parallèles

Nous avons vu dans le chapitre précédent comment augmenter le degré de parallélisme du décodeur afin d'éviter la duplication des mémoires. Dans ce chapitre nous abordons l'augmentation de l'activité des ressources de calcul afin de les utiliser au mieux. Nous avions vu dans le deuxième chapitre que cette activité décroît fortement avec l'augmentation du

degré de parallélisme. Pour y remédier, nous proposons tout d'abord un nouveau séquencement *SW-Σ** (*cf.* Figure F-8), qui utilise au mieux les ressources de calcul de l'information extrinsèque que le séquencement *SW-Σ*^{0} en les partageant entre deux processeurs, lui permettant ainsi d'atteindre une activité maximale égale à 1 (contrairement au séquencement *SW-Σ*^{0} dont l'activité est bornée par une valeur inférieure à 1). L'activité de ce nouveau séquencement est ainsi dans tous les cas est supérieure à celle du séquencement *SW-Σ*$^-$ (voir Figure F-8).

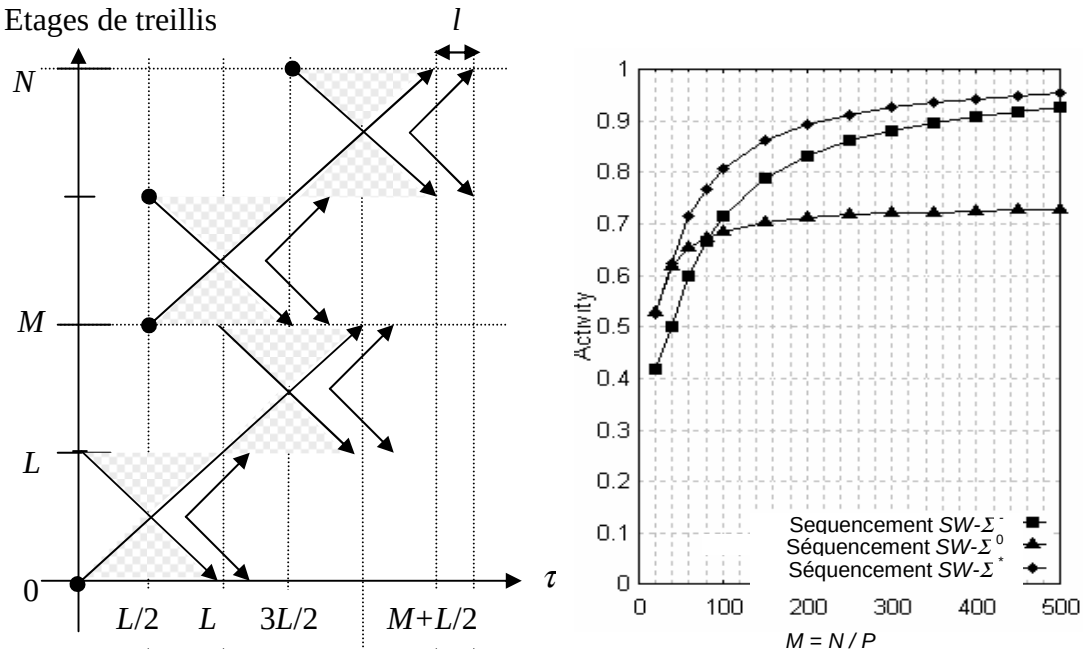


Figure F-8 : Séquencement *SW-Σ** et comparaison des activités avec les séquencements *SW-Σ*$^-$ et *SW-Σ*^{0}.

Néanmoins, avec ce nouveau séquencement, l'activité décroît toujours lorsque la taille de bloc diminue (voir Figure F-8). Pour atténuer cette decroissance, une première alternative consiste à utiliser une architecture hybride qui combine les architectures série et parallèle : décodage série pour les petits blocs, décodage parallèle pour les grands de sorte à obtenir une activité pratiquement constante quelle que soit la taille du bloc. Mais cette solution impose des contraintes systèmes importantes sur la latence. Une deuxième alternative, suppose un recouvrement de deux demi-itérations successives: au lieu d'attendre la fin d'une demi-itération avant de commencer la suivante, les unités de calcul déjà disponibles sont utilisées pour commencer la demi-itération suivante, ce qui conduit à une augmentation de l'activité. Nous montrons qu'à cause de la présence de l'entrelaceur entre les deux demi-itérations, il est possible qu'une information extrinsèque lue au début de la demi-itération n'ait pas encore été produite par la demi-itération précédente, générant ainsi ce que nous appelons un conflit de consistance. Par une évaluation théorique de la proportion de conflits de consistance pour un entrelaceur uniforme, qui varie quadratiquement avec *P*, nous montrons qu'il est nécessaire de résoudre ces conflits, de manière similaire aux conflits de parallélisme : à l'exécution, la compilation ou à la conception. Dans la suite du chapitre, nous abordons successivement les solutions à l'exécution et à la conception. La solution opérant à l'exécution utilise un

algorithme sous-optimal, mais les résultats de simulation montrent que cette solution, bien que prometteuse, ne permet pas pour l'instant d'augmenter le débit du décodeur. En effet, le gain en débit autorisé par l'augmentation de l'activité est réduit à néant par réduction de convergence due à la sous-optimalité de l'algorithme. La solution opérant à la conception consiste à définir conjointement l'entrelaceur avec l'architecture de décodage et en particulier avec le séquencement parallèle. Par construction, un tel entrelaceur garantit l'absence de conflits de consistance. Pour cette conception de l'entrelaceur, nous avons utilisé une représentation graphique bidimensionnelle de l'entrelaceur sur laquelle nous superposons le séquencement de décodage permettant de visualiser les zones de recouvrement des demi-itérations. Cette représentation définit ainsi un masque que l'entrelaceur doit satisfaire afin d'éviter la présence de conflits de consistance. Nous montrerons que cet objectif n'est pas simple à atteindre, mais que par l'utilisation des turbocodes à roulettes associé à un entrelaceur hiérarchique à trois niveaux nous obtenons de bons résultats : un gain de 0.2 dB à débit constant, ou 25 % en débit à performances égales, sur un exemple concret. Cet exemple démontre ici clairement les meilleures performances obtenues avec des turbocodes à roulettes.

F.5 Augmentation de la fréquence d'horloge

Dans ce chapitre nous présentons une méthode permettant d'augmenter la fréquence de fonctionnement du décodeur. Après la mise en évidence du chemin critique à l'intérieur du calcul récursif des métriques de nœuds (Aller ou Retour), nous montrons la difficulté qui apparaît lorsque l'on insère des registres à l'intérieur de ce calcul récursif. Nous continuons par un état de l'art détaillé des méthodes proposées dans la littérature pour résoudre ce problème, qui consiste à multiplexer temporellement plusieurs sections de treillis sur le même matériel. Puis nous décrivons la méthode que nous proposons qui consiste à multiplexer sur le même matériel les différentes sections de treillis mises en évidence dans le cas d'une architecture parallèle. Nous justifions l'intérêt de cette technique dans le cas d'un turbocode double-binaire. Après avoir exposé l'application de cette technique aux séquencements décrits dans cette thèse, nous avons montrée qu'elle permet de doubler le débit sans augmentation de la complexité du décodeur sur une cible FPGA.

F.6 Application : conception d'un turbo décodeur multiprocesseurs pour le standard WiMAX

La plupart des techniques développées dans les chapitres précédents sont appliquées dans ce dernier chapitre pour réaliser un turbo-décodeur pour le standard d'accès sans-fil large bande WiMAX (IEEE 802.16). Après une brève description de ce standard, nous justifions l'architecture choisie pour le décodeur. Les résultats de synthèse permettent de montrer que l'on peut atteindre des débits de l'ordre de 100 Mbits/s sur un seul FPGA. Nous terminons ce chapitre par des discussions sur les compromis en terme de quantification des données d'entrée, de débit, de performance de décodage et de complexité du décodeur.

Conclusion et perspectives

Depuis leur divulgation à la communauté scientifique en 1993, les turbocodes ont donné lieu à de nombreuses recherches tant sur le plan théorique que pratique afin de permettre leur implémentation avec une complexité matérielle limitée. Ce dernier objectif, qui a été atteint grâce à l'introduction du principe de décodage itératif explique en partie leur introduction rapide dans plusieurs standards de communication. La problématique de décodage à haut débit de turbocodes, objet de cette thèse, est relativement récente et s'explique par les besoins croissants dans le domaine des réseaux large bande. Nous avons montré que cette problématique nécessite une mise en œuvre de techniques algorithmiques et architecturales évoluées.

Le point de départ de notre étude des architectures de décodage à haut débit est l'algorithme Max-Log-MAP. Cet algorithme sous-optimal à entrée et sortie pondérées représente le meilleur compromis en entre la performance de décodage et la complexité. Dans un premier temps, nous avons montré comment simplifier au maximum cet algorithme afin d'en obtenir sa plus simple expression. Puis, nous avons proposé un modèle simple et général permettant d'exprimer le débit et l'efficacité (débit sur complexité) d'une architecture. Ce modèle appliqué au turbo-décodage a permis de mettre en évidence trois paramètres caractéristiques ayant un impact sur le débit et l'efficacité du décodeur : le degré de parallélisme, le taux d'utilisation (activité) des unités de calcul et la fréquence d'horloge. Sur chacun de ces trois axes de recherches, nous avons apporté des solutions qui constituent les contributions majeures de cette thèse.

Notre première contribution est relative à la résolution du problème des accès concurrents aux mémoires résultant d'une architecture parallèle de décodage, architecture dans laquelle plusieurs processeurs à entrée et sortie souples décodent en parallèle le mot de code reçu, dans le but de réduire la mémoire du décodeur. Dans ce sens, nous avons proposé un nouveau schéma de codage appelé *turbocodes à roulettes* (ou Multiple Slice Turbo Codes en anglais) associé à un entrelaceur contraint permettant de garantir l'absence de conflits d'accès aux mémoires. Cette solution qui consiste à résoudre les conflits lors de la conception du code conduit à une implantation matérielle de faible complexité par rapport aux autres solutions proposées dans la littérature, qui résolvent les accès concurrents soit par l'ajout de matériel ad-hoc permettant de résoudre les conflits lors de l'exécution, soit par le choix d'une organisation mémoire adaptée au code et à l'architecture. La conception des turbocodes à roulettes et l'optimisation de l'entrelaceur hiérarchique associé ont fait l'objet de nombreuses publications et d'un dépôt de brevet.

Dans le cadre d'une architecture parallèle, notre deuxième contribution concerne l'augmentation de l'activité des ressources de calcul des processeurs. Cette problématique mise en évidence pour la première fois dans le cadre de cette thèse est un problème majeur des architectures parallèles. Afin d'obtenir une augmentation significative de l'activité, nous avons proposé de nouveaux séquencements, tant au niveau interne des processeurs, qu'au

niveau global du processus itératif. Cette dernière solution nécessite l'utilisation d'une méthodologie de conception contrainte de l'entrelaceur conjointement avec l'architecture et le séquencement de décodage. Nous avons montré que l'utilisation des turbocodes à roulettes avec cette solution permet des gains de 0.2 dB en performance ou 25 % en débit.

Enfin, notre troisième contribution concerne la résolution du goulet d'étranglement du calcul récursif des métriques de nœuds. En introduisant des barrières de registres supplémentaire en association avec une technique de multiplexage temporel, nous avons obtenu un doublement de la fréquence d'horloge sans coût matériel additionnel sur un circuit FPGA.

La plupart des idées développées dans cette thèse ont été validées par la conception d'un turbo-décodeur, qui permet de démontrer la faisabilité d'implantations atteignant plusieurs centaines de Mbits/s sur un seul FPGA.

Le travail présenté dans cette thèse permet de mettre en évidence un axe de recherche où des gains significatifs peuvent encore être obtenus : l'augmentation de l'activité des processeurs. Les premiers résultats obtenus sur ce sujet seront approfondis et améliorées, de nouvelles voies seront explorées, en particulier pour des codes ou ensemble de codes prédéfinis. Cette thèse a permis de montrer que les gains les plus importants permettant d'obtenir le meilleur compromis entre débit et complexité ont été atteints avec une conception conjointe du code et de l'architecture ce qui permet d'envisager d'utiliser cette méthodologie pour d'autres décodeurs de codes correcteur d'erreurs (LDPC) ou d'autres systèmes décodés de manière itérative.

Bibliography

[3GPP] 3GPP Technical Specification Group, Multiplexing and channel coding (FDD), 3GPP Technical Specification, TS 25.212 v2.0.0, 1999.

[3GPP2] 3GPP2 C.S0002-A, "Physical layer Standard for CDMA2000 Spread Spectrum Systems," Release A, Version 55, December 1999.

[Arzel *et al*.-05] M. Arzel, C. Lahuec, F. Seguin, D. Gnaedig and M. Jézéquel, "Analog slice turbo decoder," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, ISCAS'05, May 2005.

[Bahl *et al*.-74] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, Vol. IT-20, pp. 284–287, March 1974.

[Battail-87] G. Battail, "Pondération des symbols décodés par l'algorithme de Viterbi," *Annales des télécommunications*, Vol. 1-2, pp. 31-38, January – February 1987.

[Battail *et al*.-79] G. Battail, M. C. Decouvelaere, and P. Godlewski, "Replication decoding," *IEEE Transactions on Information Theory*, Vol. IT-25, pp. 332–345, May 1979.

[Benedetto *et al*.-96a] S. Benedetto and G. Montorsi, "Unveiling turbo-codes: some results on parallel concatenated coding schemes," *IEEE Transactions on Information Theory*, Vol. 42, No. 2, pp. 409--429, March 1996.

[Benedetto *et al*.-96b] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes," *JPL TDA Progress Report*, Vol. 42-127, November 1996.

[Benedetto *et al*.-98] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial Concatenation of Interleaved Codes: Performance Analysis, Design and Iterative Decoding," *IEEE Transaction on Information Theory*, Vol. 44, No. 3, pp. 909-926, May 1998.

[Benedetto *et al*.-05] S. Benedetto, C. Berrou, C. Douillard, R. Garello, D. Giancristofaro, A. Ginesi, L. Giugno, M. Luise and G. Montorsi, "MHOMS: high speed ACM modem for satellite applications," to be published in *IEEE Wireless Communications Magazine*, 2005.

[Berrou *et al*.-03a] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," in *Proceedings of the IEEE International Conference on Communications*, ICC'93, Geneva, Switzerland, pp. 1064-1070, May 1993.

[Berrou *et al*.-93b] C. Berrou, P. Adde, E. Angui and S. Faudeil, "A low complexity soft-output Viterbi decoder architecture," in *Proceedings* of the *IEEE International Conference on Communications*, ICC'93, Geneva, Switzerland, pp.737-740, May 1993.

[Berrou *et al*.-96] C. Berrou and A. Glavieux, "Near Optimum Error Correcting Coding and Decoding: Turbo Codes," *IEEE Transaction on Communications*, Vol. 44, No.10, pp. 1261-1271, October 1996.

[Berrou *et al*.-97] C. Berrou and C. Douillard, "Turbo codes pour données transmises par salves(turbo codes convolutifs pour blocs courts)," *Rapport*

intermédiaire du marché d'études CCETT/ENSTB 96ME08, July 1997.

[Berrou *et al*.-99a] C. Berrou, C. Douillard and M. Jézéquel, "Multiple parallel concatenation of circular recursive systematic convolutional (CRSC) codes," *Annales des Télécommunications*, Tome 54, No. 3-4, pp. 166-172, March 1997.

[Berrou *et al*.-99b] C. Berrou, C. Douillard and M. Jézéquel, "Designing turbo codes for low error rates," in *Proceedings of IEE Colloquium : "Turbo codes in digital broadcasting - Could it double capacity?,"* London, England, November 1999.

[Berrou *et al*.-02] C. Berrou, S. Vaton, M. Jézéquel and C. Douillard, "Computing the minimum distance of linear codes by the error impulse method," in *Proceedings of the Global Telecommunications Conference*, GLOBECOM '02, Taipei, Taiwan, Vol. 2, pp. 1017- 1020, November 2002.

[Berrou *et al*.-04] C. Berrou , Y. Saouter, C. Douillard , S. Kerouedan and M. Jézéquel, "Designing good permutations for turbo codes: towards a single model," *in Proceedings of the IEEE International Conference on Communications*, Paris, France, Vol. 1, pp.341-345, June 2004.

[Blanksby *et al*.-02] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s1024-b, Rate-1/2 Low-Density Parity-Check Decoder," *IEEE Journal on Solid-State Circuits*, Vol. 37, pp. 404–412, March 2002.

[Boutillon *et al*.-00] E. Boutillon, J. Castura and R.F. Kschischang, "Decoder-first code design," in *Proceedings of the 2nd International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 459-462, September 2000.

[Boutillon et *al*.-02] E. Boutillon, P.Glenn Gulak, V. Gaudet and D. Gnaedig, "Procédé de codage et/ou de décodage de codes correcteurs d'erreurs, dispositifs et signal correspondants," Patent N° 0204764, France, April 2002.

[Boutillon *et al*.-03] E. Boutillon, W. J. Gross and P. G. Gulak, "VLSI Architectures for the MAP Algorithm," *IEEE Transactions on Communications*, Vol. 51, No. 2, pp. 175-185, February 2003.

[Boutillon *et al*.-05] E. Boutillon and D. Gnaedig, "Maximum Spread of D-dimensional Multiple Turbo Codes", to be published in *IEEE Transactions on Communications*, 2005.

[CAS-95] CAS 5093 40 Mb/s Turbo Code Decoder. December Rev 4.1. Comatlas. May 1995.

[CCSDS-99] CCSDS, Telemetry channel coding, CCSDS 101.0-B-4, Blue Book, May 1999.

[Classon *et al*.-00] B. Classon, K. Blankenship and V. Desai, "Turbo decoding with the constant-log-map algorithm," in *Proceedings of the 2nd International Symposium on Turbo codes and Related Topics*, pp. 467-470, September 2000.

[Chen *et al*.-01] J. Chen and M.P.C. Fossorier, " Near optimum universal belief propagation based decoding of LDPC codes and extension to turbo decoding," in *Proceedings of IEEE International Symposium on Information Theory*, p. 189, June 2001.

[Crozier *et al*.-00] S. Crozier, "New High-Spread High-Distance Interleavers for Turbo-Codes," in *Proceedings of the 20th biennial Symposium on Communications*, pp. 3-7, Kingston, Canada, May 2000.

[Crozier *et al*.-01] S. Crozier and P. Guinand, "High-Performance Low-Memory Interleaver Banks for Turbo-Codes," in *Proceedings of the IEEE 54th Vehicular Technology Conference*, VTC, Atlantic City, USA, pp. 2394-2398, October 2001.

[Crozier *et al*.-02] S. Crozier and P. Guinand, "Distance Bounds and the Design of High-Distance Interleavers for Turbo-Codes," in *Proceedings of the 21st Biennial Symposium on Communications*, Kingston, Canada, pp.10-14, June 2002.

[Crozier *et al*.-03] S. Crozier and P.Guinand, "Distance Upper Bounds and True Minimum Distance Results for Turbo-Codes designed with DRP Interleavers," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, pp. 169-173, September 2003.

[Dawid *et al*.-92] H. Dawid, S. Bitterlich and H. Meyr, "Trellis pipeline-interleaving: a novel method for efficient Viterbi decoder implementation," in *Proceedings of the IEEE International Symposium of Circuits and Systems*, ISCAS '92, San Diego, USA, Vol. 4, pp. 1875-1878, May 1992.

[Dingninou *et al*.-99] A. Dingninou, F. Rafaoui and C. Berrou, "Organisation de la mémoire dans un turbo décodeur utilisant l'algorithme SUB-MAP," in *Proceedings of GRETSI*, France, pp. 71–74, September 1999.

[Dingninou-01] A. Dingninou, "Implémentation de turbo code pour trame courtes," Ph.D. dissertation, Université de Bretagne Occidentale, Bretagne, France, 2001.

[Dielissen *et al*.-00] J. Dielissen and J. Huisken, "State vector reduction for initialization of sliding windows MAP," in *Proceedings of the* 2nd *International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 387–390, September 2000.

[Dobkin *et al*.-02] R. Dobkin, M. Peleg and R. Ginosar, "Parallel VLSI Architecture for MAP Turbo Decoder," in *Proceedings of the 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, PIMRC'2002, Vol. 1, pp. 384-388, September 2002.

[Dolinar *et al*.-95] S. Dolinar and D. Divsalar, "Weight Distributions for Turbo Codes Using Random and Non-random Permutations," *TDA Progress Report 42-122*, April-June 1995.

[Dolinar *et al*.-98] S. Dolinar, D. Divsalar and F. Pollara, "Code Performance as A Function of Block Size," *TDA Progress Report 42-133*, pp. 1-23, May 1998.

[Douillard *et al*.-00] C. Douillard, M. Jézéquel, C. Berrou, N. Brengarth, J. Tousch and N. Pham, "The Turbo Code Standard for DVB-RCS," in *Proceedings of the 2nd International Symposium on Turbo Codes and Related Topics*, pp. 535-538, Brest, France, September 2000.

[Douillard *et al*.-05] C. Douillard and C. Berrou, "Turbo Codes with Rate-m / (m+1) Constituent Convolutional Codes," submitted to *IEEE Transactions on Communications*, 2005.

[DVB-RCS] DVB-RCS Standard, "Interaction channel for satellite distribution systems," ETSI EN 301 790, V1.2.2, pp. 21-24, Dec 2000.

[Elias-54] P. Elias, "Error-free Coding," *IRE Transactions on Information Theory*, PGIT-4, pp. 29-37, September 1954.

[Elias-55] P. Elias, "Coding for noisy channels," *IRE Convention Record*, Vol. 4, pp. 37-47, 1955.

[Erfanian *et al*.-94] J. Erfanian, S. Pasupathy and G. Gulak, "Reduced Complexity Symbol Detectors with Parallel Structure for ISI Channels," *IEEE Transactions on Communications*, Vol. 42, No. 2/3/4, pp. 1661-1671, February/March/April 1994.

[Fedorov-53] E.S. Fedorov, "Elements of the study of figures" (in Russian, 1885) Zap. Mineralog. Obsc. (2) 21, 1-279. Reprinted by Izdat. Akad. Nauk SSSR, Moscow, 1953.

[Fettweis *et al*.-89] G. Fettweis and H. Meyr , "Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck," *IEEE Transactions on Communications*, Vol. 37, No. 8, pp.785-790, August 1989.

[Forney-70] G.D. Forney, "Convolutional codes: algebraic structure", IEEE Transactions on Information Theory, Vol. IT-16, pp.720-738, November 1970.

[Forney-73] G. D. Forney. "The Viterbi algorithm," in *Proceedings of the IEEE*, Vol. 61, No. 3, pp. 268-278, March 1973.

[Fossorier *et al*.-98] M. Fossorier, F. Burkert, S. Lin and J.Hagenauer, "On the Equivalence between SOVA and Max-Log-MAP Decodings," *IEEE Communications Letters*, Vol. 2, pp. 137-139, May 1998.

[Gallager-63] R.G. Gallager, "Low-Density Parity-Check Codes," Cambridge, MIT Press, 1963.

[Garello *et al*.-01] R. Garello, P. Pierleoni and S. Benedetto, "Computing the free distance of turbo codes and serially concatenated codes with interleavers: algorithms and applications," *IEEE Journal on Selected Areas in Communications*, Vol. 19, Issue 5, pp. 800–812, May 2001.

[Garello *et al*.-04] R. Garello and A. Vila, "The all-zero iterative decoding algorithm for turbo code minimum distance computation," in *Proceedings of the IEEE International Conference on Communications*, ICC'04, Paris, France, June 2004.

[Giulietti *et al*.-02a] A. Gulietti, L. van der Perre and M. Strum, "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," *Electronics Letters*, Vol. 38, No. 5, pp. 232-234, February 2002.

[Giulietti *et al*.-02b] A. Gulietti, 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 *Proceedings of the IEEE Custom Integrated Circuits Conference*, CICC'02, Orlando, USA, pp. 389-392, May 2002.

[Gnaedig *et al*.-03] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel and P. G. Gulak, "On Multiple Slice Turbo Codes," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, Brest, pp.153-157, September 2003.

[Gnaedig *et al*.-04] D. Gnaedig, M. Lapeyre, F. Mouchoux and E. Boutillon, "Efficient SIMD technique with parallel Max-Log-MAP algorithm for turbo decoders," in *Proceedings of the Global Signal Processing Conference*, GSPx'04, Santa Clara, USA, September 2004.

[Gnaedig *et al*.-05a] D. Gnaedig, E. Boutillon, V. C. Gaudet, M. Jézéquel and P. G. Gulak, "On Multiple Slice Turbo Codes," *Annales des Télécommunications*, Vol. 60, No. 1-2, January 2005.

[Gnaedig *et al*.-05b] D. Gnaedig, E. Boutillon and M. Jézéquel, "Design of Three-Dimensional Multiple Slice Turbo Codes," in *the special issue on Turbo Processing of the EURASIP Journal on Applied Signal Processing*, Vol. 6, pp. 808-819, 2005.

[Hagenauer-88]* J. Hagenauer, "High-rate punctured convolutional codes for soft decision Viterbi decoding," *IEEE Transactions on Communications*, Vol. COM-36, No. 4, pp. 389-400, April 1988.

[Hagenhauer *et al*.-89] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft decision outputs and its applications," in *Proceedings of the Global Telecommunications Conference*, GLOBECOM'89, Dallas, Texas, pp. 47.11-47.17, November 1989.

[Hagenauer *et al*.-96] J. Hagenauer, E. Offer and L. Papke, "Iterative decoding of binary block and convolutionnal codes," *IEEE Transactions on Information Theory*, Vol. 42, pp. 429-445, 1996.

[Hartmann *et al*.-76] C. R. P. Hartmann and L. D. Rudolph, "An optimum symbol-by-symbol decoding rule for linear codes," *IEEE Transactions on Information Theory*, Vol. IT-22, pp. 514–517, 1976.

[Heegard *et al*.-99] C. Heegard and S.B. Wicker, Turbo Coding, K*luwer Academic Publishers*, 1999.

[Hekstra-89] A.P. Hekstra, "An alternative to metric rescaling in Viterbi decoders", *IEEE Transactions on Communications*, Vol. 37, pp. 1120-1222, November 1989.

[Hokfelt *et al*.-99] J. Hokfelt, O.Edfors and T. Maseng, "Interleaver Design for Turbo Codes Based on the Performance of Iterative Decoding," in *Proceeding of the IEEE International Conference on Communications*, ICC'99, Vancouver, Canada, Vol. 1, pp.93-97, 1999.

[Hsu *et al*.-98] J-M Hsu and C-L Wang, "A parallel decoding scheme for Turbo Codes," in *Proceedings of the IEEE International Symposium of Circuits and Systems*, ISCAS'98, Monterrey, USA, Vol. 4, pp. 445-448, June 1998.

[Hsu *et al*.-99] J. M. Hsu and C. L. Wang, "On finite-precision implementation of a decoder for turbo-codes," in *Proceedings of the IEEE International Symposium on Circuit and Applications*, Orlando, USA, Vol. 4, pp. 423–426, May 1999.

[Huffman *et al*.-03] W. C. Huffman and V. Pless, "Fundamentals of Error Correcting Codes," Cambridge University Press, 2003.

[Jézéquel *et al*.-97] M. Jézéquel, C. Berrou, C. Douillard and P. Pénard, "Characteristics of a sixteen-state turbo-encoder/decoder," in *Proceedings* of the *1st International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 280-283, September 1997.

[Kahale *et al*.-98] N. Kahale and R. Urbanke, "On the minimum distance of parallel and serially concatenated codes," in *Proceedings of the IEEE International Symposium on Information Theory*, ISIT'98, Cambridge, USA, August 1998.

[Kwak *et al*.-03] J. Kwak, S. Park, S. Yoon, K. Lee, "Implementation of a parallel turbo decoder with dividable interleaver," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, ISCAS '03, Bangkok, Thailand, Vol. 2, pp. 65-68, May 2003.

[Leighton-92] F. T. Leighton, "Introduction to parallel algorithms and architectures: arrays, trees and hypercubes," San Mateo, USA, *Morgan Kaufmann Publishers*, 1992.

[Lin *et al*.-89] H.D. Lin and D. G. Messerschmitt, "Algorithms and architectures for concurrent Viterbi decoding," in *Proceedings of the IEEE*

International Conference on Communications, ICC'89, Boston, USA, pp. 836-840, June 1989.

[Ma *et al*.-86] H. H. Ma and J. K. Wolf, "On Tail Biting Convolutional Codes," *IEEE Transactions on Communications*, No. 2, pp. 104-111, February 1986.

[MacKay *et al*.-96] D.J.C MacKay and R.M Neal, "Near Shannon Limit Performance of Low-Density Parity-Check Codes," *Electronics Letters*, Vol. 32, pp. 1645-1646, August 1996.

[Martinez *et al*.-03] A. Martinez and M. Rovini, "Iterative Decoders Based on Statistical Multiplexing," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 423-426, September 2003.

[Masera *et al*.-99] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Transactions on VLSI Systems*, Vol. 7, pp. 369–379, September 1999.

[Montorsi *et al*.-01a] G. Montorsi and S. Benedetto, "Design of fixed-point iterative decoders for concatenated codes with interleavers," *IEEE Journal Selected Areas in Commuications*, Vol. 19, pp. 871-882, May 2001.

[Montorsi *et al*.-01b] G. Montorsi and S. Benedetto, "An additive Version of the SISO Algorithm for the Dual Code," in *Proceedings of the IEEE International Symposium on Information Theory*, ISIT'01, Washington, USA, p. 27, June 2001.

[Nimbalker *et al*.-03] A. Nimbalker, T. K. Blankenship, B. Classon, T. E. Fuja and D. J. Costello, *"Inter-Window Shuffle Interleavers for High Throughput Turbo Decoding,"* in *Proceedings of 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, September 2003.

[Nimbalker *et al*.-04] A. Nimbalker, T. K. Blankenship, B. Classon, T. E. Fuja and D. J. Costello, Jr, "Contention-free interleavers," in *Proceedings of the IEEE International Symposium on Information Theory*, ISIT'04, Chicago, USA, June 2004.

[Park *et al*.-00] G. Park, S.Yoon, C. Kang and D. Hong, "An Implementation Method of a Turbo-code Decoder using a Block-wise MAP Algorithm," in *Proceedings of the IEEE Vehicular Technology Conference*, VTC Fall 2000, Boston, USA, pp. 2956-2961, September 2000.

[Prescher *et al*.-05] G. Prescher, T. Gemmeke and T.G. Noll, "A Parametrizable Low-Power High-Throughput Turbo-Decoder," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'05, Philadelphia, USA, pp. 25-28, March 2005.

[Proakis] J.G.Proakis, Digital Communications, 4th Edition, McGraw-Hill.

[Perez *et al*.-96] L. C. Perez, J. Seghers, and D. J. Costello, Jr., "A distance spectrum interpretation of turbo codes," *IEEE Transactions on Information Theory*, Vol. 42, pp. 1698-1709, November 1996.

[Pietrobon-00] S. Pietrobon, "Super Codes: A Flexible Multi-Rate Coding System," in *Proceedings of the 2nd International Symposium on Turbo-Codes & Related Topics*, Brest, France, pp. 141-148, September 2000.

[Pyndiah *et al*.-94] R. Pyndiah, A. Glavieux, A. Picart and S. Jacq, "Near optimum decoding of product codes," in *Proceedings of the Global Telecommunication Conference*, GLOBECOM'94, San Francisco, USA, November 1994.

[Pyndiah-98] R. Pyndiah, "Near-Optimum Decoding of Product Codes: Block

Turbo Codes," *IEEE Transactions on Communications*, Vol. 46, No. 8, pp. 1003-1010, August 1998.

[Riedel-98] S. Riedel, "Symbol-by-Symbol Map decoding algorithm for high-rate convolutional codes that use reciprocal dual codes," *IEEE Journal on Selected Areas in Communications*, Vol. 16, No. 2, pp. 175 -185, February 1998.

[Robertson *et al*.-95] P. Robertson, E. Villebrun and P. Hoeher, "A Comparison of Optimal and Sub-optimal Decoding Algorithm in the Log Domain," in *Proceedings of the IEEE International Conference on Communications*, ICC'95, Seattle, USA, pp. 1009-1013, June 1995.

[Robertson *et al*.-97] P. Robertson, P. Hoeher and E. Villebrun, "Optimal and suboptimal maximum a posteriori algorithms suitable for turbo decoding," *European Transactions on Telecommunications*, Vol. 8, pp. 119-125, March/April 1997.

[Saouter *et al*.-02a] Y. Saouter and C. Berrou, "Fast soft-output Viterbi decoding for duo-binary turbo codes," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, ISCAS '02, Phoenix, USA, Vol. 1, pp. 885-888, May 2002.

[Saouter *et al*.-02b] Y. Saouter and C. Berrou, "Fast SUBMAP decoders for duo-binary turbo-codes," in *Proceedings of the 1st IEEE International Conference on Circuits and Systems for Communications*, ICCSC'02, St Petersburg, Russia, pp. 150-153, June 2002.

[Saouter *et al*.-03] Y. Saouter, "Decoding M-binary turbo codes by the dual method," in *Proceedings of the IEEE Information Theory Workshop*, Paris, France, April 2003.

[Sawaya *et al*.-03] H. E. Sawaya and J. J. Boutros, "Irregular turbo codes with symbol-based iterative decoding," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 407-410, September 2003.

[Shannon *et al*.-48] C.E. Shannon and W. Weaver, "A mathematical theory of communication," *Bell System Technical Journal*, Vol. 27, June 1948.

[Shibutani *et al*.-99] A. Shibutani, H. Suda and F. Adachi, "Reducing average number of turbo decoding iterations," *Electronics Letters*, Vol 35, No. 9, pp. 701-702, April 1999.

[Shao *et al*.-99] R. Y. Shao, S. Lin and M. P. C. Fossorier, *"Two simple stopping criteria for turbo decoding," IEEE Transactions on Commun*ications, Vol COM-47, pp. 1117-1120, August 1999.

[Shung *et al*.-90] C. B. Shung, P. H. Siegel, G. Ungerboeck and H. K. Thapar, "VLSI architectures for metric normalization in the Viterbi algorithm" in *Proceedings* of the *IEEE International Conference on Communications*, ICC '90, Atlante, USA, Vol. 4, pp. 1723–1728, April 1990.

[Tanner-81] R.M. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions On Information Theory*, Vol. IT-27, September 1981.

[Tarable *et al*.-03] A. Tarable, G. Montorsi and S. Benedetto, "Mapping interleaving laws to parallel Turbo decoder architectures," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, pp. 153-157, September 2003.

[Tarable *et al*.-04a] A. Tarable and S. Benedetto, "Mapping interleaving laws to parallel turbo decoder architectures," *IEEE Communication Letters*, Vol. 8,

pp.162-164, March 2004.

[Tarable *et al*.-04b] A. Tarable, S. Benedetto and G. Montorsi, "Mapping interleaving Laws to Parallel Turbo and LDPC decoder Architectures," *IEEE Transactions on Information Theory*, Vol. 50, No. 9, September 2004.

[TC1000] TC1000. – TC1000 DVB-RCS Turbo Decoder Datasheet. TurboConcept (http://www.turboconcept.com).

[Thul *et al*.-02] M.J. Thul, F. Gilbert and N. Wehn, "Optimized concurrent interleaving architecture for high-throughput turbo-decoding," in *Proceedings of the International Conference on Electronics Circuits and Systems*, Dubrovnik, Croatia, pp. 1099-1102, 2002.

[Thul *et al*.-03] M.J. Thul, F. Gilbert and N. Wehn, "Concurrent Interleaving architectures for high-throughput channel coding," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'03, Hong Kong, Vol. 2, pp. 613-616, April 2003.

[Thruhachev *et al*.-01] D.V Thruhachev, M. Lentmaier and K. Sh. Zigangirov, "Some Results Concerning Design and Decoding of Turbo-Codes," in *Problems of Information Transmission*, Vol. 37, No. 3, pp. 190-205, July 2001.

[Tortellier *et al*.-90] P. Tortellier and D. Duponteil, "Dynamique des métriques de noeuds dans l'algorithme de Viterbi," *Annales des Télécommunications*, Vol. 45, no. 7-8, pp. 377-383, 1990.

[Tousch *et al*.-04] J. Tousch, E. Boutillon and D. Gnaedig, "Procédé et dispositif de décodage de codes par propagation de messages conjoints et systèmes les mettant en oeuvre," filed french patent N°0406687000, TurboConcept, France, June 17th 2004.

[Ullman] J. D. Ullman, "Computational Aspects of VLSI", *Computer Science Press*, 1984.

[Viterbi-98] A.J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, Vol. 16, pp. 260-264, February 1998.

[Vogt *et al*.-00] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *Electronics Letters*, Vol. 36, No. 23, pp. 1937-39, November 2000.

[WiMAX] WiMAX standard. IEEE P802.16-REVd/D5, May 2004 – Local and Metropolitan Area Network Part 16.

[Wu *et al*.-99] Y.Wu and B. D.Woerner, "The influence of quantization and fixed point arithmetic upon the BER performance of turbo codes," in *Proceedings IEEE Vehicular Technology Conference*, VTC'99, Houston, USA, Vol. 2, pp. 1683–1687, May 1999.

[Wu *et al*.-00] Y. Wu and B.D. Worner, "A simple stopping criterion for turbo decoding", *IEEE Communications Letters*, Vol. 4, No. 8, pp. 258-260, August 2000.

[Wan *et al*.-03] K. Wan, Q.Chen and P.Fan, "A novel parallel turbo coding technique based on frame split and trellis terminating," in *Proceedings of the 4th international Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'2003, Chengdu, China, pp. 927-930, August 2003.

[Wang *et al*.-01] Z. Wang, Z. Chi and K.K. Parhi, "Area-Efficient High Speed Decoding Schemes for Turbo/MAP Decoders," in *Proceedings of the*

IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'01, Salt Lake City, USA, Vol. 4, pp. 2633-2636, May 2001.

[Wang *et al*.-02] Z. Wang, Z. Chi and K.K. Parhi, "Area-Efficient High Speed Decoding Schemes for Turbo/MAP Decoders," *IEEE Transactions on VLSI Systems*, Vol. 10, No. 12, December 2002.

[Worm *et al*.-01] A. Worm, H. Lamm and N. Wehn, "VLSI architectures for high-speed MAP decoders," in *Proceedings of the 14th International Conference on VLSI Design*, Bangalore, India, pp. 446-453, March 2001.

[Weiß *et al*.-01] C. Weiß, C. Bettstetter, and S. Riedel, "Code Construction and Decoding of Parallel Concatenated Tail-Biting Codes," *IEEE Transactions on Information Theory*, Vol. 47, No. 1, pp. 368-388, January 2001.

[Yoon *et al*.-02] S. Yoon, Y. Bar-Ness, "A Parallel MAP Algorithm for Low Latency Turbo Decoding," *IEEE Communications Letters*, Vol. 6, No. 7, pp. 288-290, July 2002.

[Zhang *et al*.-01] T. Zhang, Z. Wang and K. K. Parhi, "On Finite Precision Implementation of Low Density Parity Check Codes Decoder," in *Proceedings of IEEE International Symposium of Circuits and Systems*, ISCAS'01, Sydney, Australia, pp. 202-205, May 2001.