Memory Efficient Implementation of the BCJR Algorithm

Simon Huettinger, Marco Breiling, Johannes Huber

Lehrstuhl für Nachrichtentechnik II, Universität Erlangen-Nürnberg
Cauerstr. 7, D-91058 Erlangen, Germany
Phone: +49-9131-852 [7116, 7668], Fax: +49-9131-852-8919,
Email: [huettinger, breiling]@LNT.de

Abstract: *In this paper, a modification of the BCJR algorithm – the Forward BCJR algorithm – is introduced. This novel algorithm can strongly reduce the memory requirements in exchange for a moderately increased computational complexity.*

Keywords: MAP decoding, Soft–In Soft–Out decoding.

1. INTRODUCTION

Since 1993, when Berrou et al. published a paper describing a new coding scheme called "turbo-codes" [1], the optimal algorithm for decoding of convolutional codes – the BCJR algorithm [2] – attracts high attention.

Various modifications of the BCJR algorithm have been proposed to achieve a reduced computational complexity [3], [4] or a fixed decoding delay [5], [6]. A small decoding delay also reduces the memory requirements, but as the delay cannot be made arbitrarily small without an overproportional increase in complexity, the memory efficiency of BCJR decoding is still limited.

In this paper, we review the development of the Sliding Window BCJR algorithm [5] in Section 2 and in Section 3 we introduce a new modification, the Forward BCJR, to reduce the memory requirements significantly. In Section 4 both algorithms are combined because the Forward BCJR can only cope with short blocks and the Sliding Window BCJR subdivides the codeword into such blocks. So a very memory efficient algorithm can be constructed that preserves the advantages of both underlying algorithms. In Section 5, a metric rescaling scheme in the log–domain is discussed. Complexity and quantization issues are addressed in Section 6.

2. SLIDING WINDOW BCJR

The optimal decoding algorithm for convolutional codes to minimize the symbol error rate and produce a posteriori probabilities (APP) was presented by Bahl et al. in 1974 [2]. This algorithm estimates the state and state transition probabilities of a binary convolutional code with L generator polynomials and Z states observed through a discrete memoryless channel. The channel output $\boldsymbol{y}_k = (y_{k,1} \dots y_{k,L})$ corresponds to a single state transition.

The BCJR algorithm recursively calculates the scaled probability distribution $\boldsymbol{\beta}_k = (\beta_k(1) \dots \beta_k(Z))$ within a block of length N processing the received sequence backward from trellis segment $k = N$ to $k = 0$. $\beta_k(m)$ is proportional to the probability of being in state $m \in \{1 \dots Z\}$ at trellis segment k having observed the channel output $\boldsymbol{y}_{k+1} \dots \boldsymbol{y}_N$. By the use of the state transition probabilities $\gamma_k(m, m')$, that describe the probability of the encoder being in state $m' \in \{1 \dots Z\}$ at trellis segment k having observed $\boldsymbol{y}_k$ under the condition that the encoder state was m for trellis segment $k-1$, $\beta_k(m)$ is calculated recursively within the following equation:

$$\beta_k(m) \quad = \quad \sum_{m'} \beta_{k+1}(m') \cdot \gamma_{k+1}(m, m') \qquad (1)$$

In a forward recursion the scaled probability distribution $\boldsymbol{\alpha}_k = (\alpha_k(1) \dots \alpha_k(Z))$ that depends on $\boldsymbol{y}_1 \dots \boldsymbol{y}_k$ is determined recursively starting from the beginning of the received sequence.

$$\alpha_k(m') \quad = \quad \sum_{m} \alpha_{k-1}(m) \cdot \gamma_k(m, m') \qquad (2)$$

Each state transition probability $\gamma_k(m, m')$ is uniquely connected to an information symbol $x \in \{0, 1\}$. Hence we can mark all state transitions that belong to the same symbol by $\gamma_k^x(m, m')$. Then the APP for trellis segment k is calculated from $\boldsymbol{\alpha}_k$, $\boldsymbol{\beta}_k$ and $\boldsymbol{\gamma}_k^x$, cf. [5]:

$$\Pr(X_k{=}x) \; = \; c \sum_{m} \sum_{m'} \alpha_k(m) \gamma_k^x(m, m') \beta_{k+1}(m') \qquad (3)$$

To obtain probabilities from Eq. (3) we have to choose c such that $\Pr(X_k = 1) + \Pr(X_k = 0) = 1$.

The complete backward recursion has to be performed and the $\boldsymbol{\beta}$ values have to be stored before the first APP can be calculated. Hence, the memory requirements grow *linearly* with the block length N. To avoid this effect the Sliding Window BCJR algorithm [5] can be used. It divides the received sequence into smaller blocks – the windows – which can be processed sequentially. Within a window the backward recursion is started from an arbitrary, e.g. an uniform distribution as substitution for the unknown

$\boldsymbol{\beta}_N$. Then the algorithm is processed for l_ν trellis segments until the $\boldsymbol{\beta}$ values have sufficiently converged towards the correct state probabilities. Within these l_ν trellis segments, that belong to the so–called *overlap zone*, no APP output is generated and hence, they have to be included in more than one window. If l_ν is larger than five times the constraint length ν of the convolutional code the loss due to starting from an uniform instead of the correct state probability distribution is negligible [5]. Often even lower values of l_ν prove to be sufficient.

By changing the size of the window, memory requirements can be traded against complexity to a certain extent. A favorable solution results when l_ν is chosen half the window size, as a forward and two backward recursions (cf. Fig. 2.) can be performed in parallel at the original symbol clock, simplifying integrated circuit (IC) design significantly.

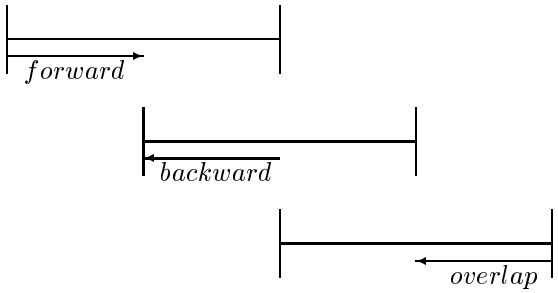

Figure 1: Sliding Window BCJR algorithm for parallel implementation.

3. FORWARD BCJR

The recursive formula of Eq. (1) can be expressed as a matrix multiplication with a sparse matrix $\boldsymbol{\Gamma}_k$ with elements $\boldsymbol{\gamma}_k(m, m')$ where $m, m' \in \{1 \ldots Z\}$:

$$\boldsymbol{\beta}_k \quad = \quad \boldsymbol{\Gamma}_{k+1} \cdot \boldsymbol{\beta}_{k+1} \tag{4}$$

From Eq. (4) it is obvious that the backward recursion can be reversed if $\boldsymbol{\Gamma}_k$ is not singular

$$\boldsymbol{\beta}_k \quad = \quad \boldsymbol{\Gamma}_k^{-1} \cdot \boldsymbol{\beta}_{k-1}. \tag{5}$$

To simplify the matrix inversion we can use its structural properties and decompose $\boldsymbol{\Gamma}_k^{-1}$ into a constant c_Γ that can be neglected since the APP output is invariant to constant factors, the matrix $\boldsymbol{\Lambda}_k$ that can be found by simply reordering the elements of $\boldsymbol{\Gamma}_k$, and a diagonal matrix $\boldsymbol{\Delta}_k = \mathrm{diag}(1 \;\; c_\Delta \;\; 1 \;\; c_\Delta \;\; \ldots)$. Hence the computation of $\boldsymbol{\Gamma}_k^{-1}$ is of low complexity using this decomposition

$$\boldsymbol{\Gamma}_k^{-1} \quad = \quad c_\Gamma \cdot \boldsymbol{\Lambda}_k \cdot \boldsymbol{\Delta}_k. \tag{6}$$

Due to the fact that the proportion of the elements of $\boldsymbol{\Gamma}_k^{-1}$ that are zero grows very fast with Z, the calculation of $\boldsymbol{\Delta}_k$ or equivalently the calculation of c_{diag} is independent of the number of states Z and hence of fixed complexity.

In Eqs. (7), (8), and (9) (see bottom of this page) the decomposition is given for the worst case of a rate 1/2 convolutional code with $Z = 4$ states, where the matrix $\boldsymbol{\Gamma}_k^{-1}$ is densest.

Singularity of $\boldsymbol{\Gamma}_k$ occurs if two branches in the trellis, which are assigned to different code symbols, have the same metric. This case can be avoided if it is ensured that the members of $\boldsymbol{y}_k$ are not equal ($y_{k,1} \neq y_{k,2}$). In a practical implementation this can be easily ensured by setting the least significant bits of $y_{k,1}$ and $y_{k,2}$ to fixed different values. However, this is equal to additional weak noise and slightly degrades the performance of the algorithm. Note that only the signal for the backward recursion is corrupted by this deterministic artificial noise. The forward recursion still can work on the original received signal.

The new algorithm processes the complete backward recursion for the whole codeword starting from $\boldsymbol{\beta}_N$ to obtain the probability distribution $\boldsymbol{\beta}_0$ *without storing intermediate results*. Then in a forward recursion the distributions $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ can be calculated simultaneously. At the same time the APP can be determined. As memory has to be reserved only for the current $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$, this is a very memory efficient version of the BCJR algorithm.

Unfortunately this algorithm may be numerically instable if N becomes large. A solution is to store the $\boldsymbol{\beta}_k$ values during the backward recursion every d

$$\boldsymbol{\Gamma}_k \quad = \quad \begin{pmatrix} y_{k,1}y_{k,2} & 0 & (1-y_{k,1})(1-y_{k,2}) & 0 \\ (1-y_{k,1})(1-y_{k,2}) & 0 & y_{k,1}y_{k,2} & 0 \\ 0 & y_{k,1}(1-y_{k,2}) & 0 & (1-y_{k,1})y_{k,2} \\ 0 & (1-y_{k,1})y_{k,2} & 0 & y_{k,1}(1-y_{k,2}) \end{pmatrix} \tag{7}$$

$$\boldsymbol{\Lambda}_k \quad = \quad \begin{pmatrix} y_{k,1}y_{k,2} & -(1-y_{k,1})(1-y_{k,2}) & 0 & 0 \\ 0 & 0 & y_{k,1}(1-y_{k,2}) & -(1-y_{k,1})y_{k,2} \\ -(1-y_{k,1})(1-y_{k,2}) & y_{k,1}y_{k,2} & 0 & 0 \\ 0 & 0 & -(1-y_{k,1})y_{k,2} & y_{k,1}(1-y_{k,2}) \end{pmatrix} \tag{8}$$

$$c_\Delta \quad = \quad \frac{(y_{k,1}+y_{k,2}-1)(1-y_{k,1}-y_{k,2}-2y_{k,1}y_{k,2})}{(y_{k,1}-y_{k,2})(y_{k,1}+y_{k,2}-2y_{k,1}y_{k,2})} \tag{9}$$

trellis segments. In this case the forward calculation of $\boldsymbol{\beta}_k$ can be reset to the stored values every d trellis segments. This leads to an algorithm that performs nearly optimal and reduces the memory requirements significantly especially for convolutional codes with short block length and large constraint length, if d is chosen appropriately (e.g. ten times constraint length). This is exactly the inverse behavior compared to the Sliding Window BCJR algorithm.

4. COMBINED FORWARD / SLIDING WINDOW BCJR

It is obvious that a combination of the Forward and the Sliding Window BCJR is a very powerful algorithm regarding memory requirements.

If a Sliding Window BCJR algorithm with window size 10ν and overlap zone length $l_\nu = 5\nu$ is used, in every window a Forward BCJR algorithm without storing intermediate results can be performed with negligible performance degradation, as the limiting conditions – the minimal overlap zone length l_ν and the maximal block length d – are satisfied.

The algorithm now only needs working memory for that values which are involved in every calculation and has to load only the channel output $\boldsymbol{y}_k$ that determines $\boldsymbol{\gamma}_k$ from the memory. E.g. for decoding of a rate 1/2 code with memory 2 a state–of–the–art digital signal processor (DSP) comprises enough registers to store the quantized $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ values. Access to these values is even faster than the on–chip memory. Furthermore, in every step only two values $y_{k,1}, y_{k,2}$ have to be loaded. For small to medium block length it is possible to abandon the use of extern memory, as the whole received sequence can be stored in the on–chip memory. This will certainly improve the speed of operation. Hence, this algorithm will yield a speed gain in spite of the increased computational complexity.

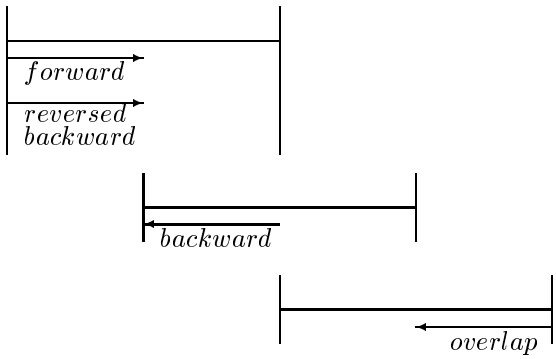

Figure 2: Combined Forward / Sliding Window BCJR algorithm for parallel implementation in symbol clock

5. METRIC RESCALING IN THE LOG DOMAIN

For implementation all variables have to be of finite range. Hence, the $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ values, that represent unboundedly growing accumulated path differences have to be rescaled within every recursion step.

A very attractive rescaling scheme was introduced for Viterbi decoders in [7]. Fortunately this method is also applicable in a straightforward way to the BCJR algorithm. In the log–domain the difference between metrics is bounded as long as the branch metrics are bounded. The branch metrics that consist of the channel output vector $\boldsymbol{y}_k$ are surely bounded as they are also quantized variables. If the range for the $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ difference values is larger than $2 \cdot \nu \cdot \max(\boldsymbol{y}_k)$, an overflow does not occur.

In practical implementations this range is quite large and the granularity becomes coarse for a finite number of bits spent for each $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ difference value. Hence, there is a trade–off between exactness and probability of overflow. Simulations have shown that it is quite efficient to choose a range for the $\boldsymbol{\beta}_k$ values which avoids any overflow, whereas the $\boldsymbol{\alpha}_k$ values should be quantized very fine within a smaller total range.

6. COMPLEXITY AND QUANTIZATION

The speed of operation of any implementation is determined by the number of operations that have to be performed by the algorithm. But extern memory access can slow down especially highly pipelined and parallelized DSP stuctures dramatically. In [8] it has been shown that there is a speed gain for the Sliding Window BCJR algorithm compared to the original algorithm of about a factor of *five*, if the block length exceeds 1000-information bits, although the Sliding Window BCJR algorithm requires more operations. This was due to the problem that for the conventional BCJR algorithm the $\boldsymbol{\beta}_k$ values had to be swapped into extern memory. With the combined Forward / Sliding Window BCJR algorithm swapping can even be avoided for the channel output symbols for moderate block lengths. Hence, despite the further increased number of operations an additional speed gain can be expected.

The computational complexity for decoding a rate 1/2 code can be given by analyzing the algorithm's equations. The total number of operations needed to decode a single bit is given in Table 1 for a code with $Z = 16$ states. For implementation in the log–domain each multiplication equals one addition and each addition of probabilities needs one

max–operation, one table look–up, and one addition of log–likelihood ratios [9].

The forward recursion Eq. (2) as well as the backward recursion Eq. (1) need one addition and two multiplications for each state m, as only for two values of m respectively m' the probabilities $\gamma_k(m, m')$ are nonzero. Metric rescaling as described in Section 5 is done by a single multiplication for each $\boldsymbol{\alpha}_k$ or $\boldsymbol{\beta}_k$ value. The calculation of APPs in Eq. (3) 'costs' one addition and two multiplications for each state.

The additional complexity introduced by the sliding window extension with an overlap zone of half the window size is that of an additional backward recursion and hence one addition and two multiplications per state. The forward algorithm adds the fixed complexity of the calculation of c_Δ that can be done with four additions and four multiplications, if the intermediate results are stored. Furthermore, the matrix multiplication $\boldsymbol{\Delta}_k \cdot \boldsymbol{\beta}_k$ has to be performed. It comprises only one multiplication and one addition per state, as every second element of the main diagonal of $\boldsymbol{\Delta}_k$ is 1.

Table 1: Computational Complexity
(Operations per Trellis Segment)

	Multiplications	Additions
BCJR	144	48
Sliding Window	192	64
Forward	164	68
Combined	228	100

The quantization ranges that have to be chosen to achieve a close to optimum performance are a very crucial point because the absolute memory requirements depend on it very strongly. Hence, for the standard BCJR and the Sliding Window algorithm the number of levels for the $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ values has to be small. However, due to the large variation that are the result of the iterative decoding process in a turbo decoder the range has to be large enough. The ranges and quantization granularities of [3] (see also Table 2) are a very good trade–off between performance and memory efficiency for implementation in the log–domain.

Table 2: Quantization
(for BCJR and Sliding Window BCJR algorithm)

Variable	Number of Levels	Limit Values	
$\beta_k(m)$	256	-127	128
$\alpha_k(m)$	256	-15.875	16
$\gamma_k(m, m')$	256	-15.875	16
$\Pr(X_k{=}x)$	256	-31.75	32
$y_{k,1}, y_{k,2}$	64	-1.9375	2

Within the Forward or the Combined Forward / Sliding Window algorithm the situation changes, as especially for the $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ values no memory has to be reserved. Hence, a quantization of $\boldsymbol{\alpha}_k$, $\boldsymbol{\beta}_k$, and $\gamma(m, m')$ with a granularity of 16 bits – the register size of the DSP – is reasonable even from a memory efficiency point of view, as only temporary values have increased memory requirements.

Such a fine quantization of $\boldsymbol{\beta}_k$ and $\gamma(m, m')$ is even mandatory for a good performance. It minimizes the artificially introduced noise that is required to avoid singularity of $\boldsymbol{\Gamma}_k$. Furthermore, it ensures that within the window length of the combined Forward / Sliding Window algorithm the re–calculated $\boldsymbol{\beta}_k$ values are close enough to the original ones, to avoid practically any performance degradation.

The quantization of $\boldsymbol{\alpha}_k$ can be chosen to be 8 or 16 bits, but to ease the implementation it is reasonable to use the same quantization as for the $\boldsymbol{\beta}_k$ values.

REFERENCES

[1] Berrou, C., Glavieux, A., Thitimajshima, P., "Near Shannon Limit Error–Correcting Coding and Decoding: Turbo–Codes". *Proc. of ICC '93*, pp. 1064-1070, 1993.

[2] Bahl, L.,Cocke, J., Jelinek, F., Raviv, J., "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate". *IEEE Transactions on Information Theory*, pp. 284–287, 1974.

[3] Robertson, P., Villebrun, E., Hoeher, P., "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain" *Proc. of ICC '95*, Seattle, WA, USA, June 1995, pp. 1009-1013, 1995.

[4] Pietrobon, S., Barbulescu, S., "A simplification of the modified Bahl decoding algorithm for systematic convolutional codes", *International Symposium on Information Theory and its Applications*, Sydney, NSW, pp. 1073-1077, Nov. 1994.

[5] Huber, J., *Trelliscodierung* [in German], pp. 126–134, Springer-Verlag, Berlin, 1992.

[6] Benedetto, S., Divsalar, D., Montorsi, G., Pollard, F., "Soft-Output Decoding Algorithms for Continuous Decoding of Parallel Concatenated Convolutional Codes" *Proc. of ICC '96*, Dallas, Texas, June 1996, 1996.

[7] Hekstra, A., "An Alternative to Metric Rescaling in Viterbi Decoders", *IEEE Transactions on Communications*, vol. 37, no. 11, pp. 1220-1221, Nov. 1989.

[8] Zottmann, A., "Implementierung digitaler Übertragungssysteme mit Turbo-Codes auf Signalprozessoren" [in German], Master Thesis, University Erlangen-Nuremburg, Jun. 2000.

[9] Petersen, J., "Implementierungsaspekte zur Symbol-by-Symbol MAP-Dekodierung von Faltungscodes" [in German], *ITG Fachtagung, Codierung für Quelle, Kanal und Übertrager*, 1994, pp. 41–48.