VLSI Architectures for the Forward-Backward Algorithm

Emmanuel Boutillon[1], Warren J. Gross[2], and Glenn Gulak[2]

[1] Département COMELEC
Ecole Nationale Supérieure des Télécomunications (ENST)
46 rue Barrault, 75634 Paris cedex 13, France
[2] Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4, Canada

Submitted as a transaction paper to the IEEE Transactions on Communications

Abstract: This paper presents techniques for the VLSI implementation of the forward-backward algorithm, sometimes known as the MAP or BCJR algorithm. In general, knowledge about the implementation of the Viterbi algorithm can be applied to the forward-backward algorithm. Bounds are derived for the dynamic range of the state metrics which enable the designer to optimize the word length. The computational kernel of the algorithm is the Add-MAX* operation, which is the Add-Compare-Select (ACS) operation of the Viterbi algorithm with an added Offset (ACSO). We show that the critical path of the algorithm can be reduced if the Add-MAX* operation is reordered into an Offset-Add-Compare-Select (OACS) operation by adjusting the location of registers. A general architecture for the forward-backward algorithm is presented which exposes trade-offs between computational complexity, latency and memory size. Some of these architectures eliminate the need for RAM blocks with unusual form-factors or can replace the RAM with registers. These architectures are suited to VLSI implementation of turbo decoders.

Index Terms - Forward-backward algorithm, MAP estimation, Turbo codes, Viterbi decoding, Very-large-scale integration

I. Introduction

In recent years, there has been considerable interest in soft-output decoding algorithms; algorithms that provide a measure of reliability for each bit that they decode. Perhaps the most promising application of soft-output decoding algorithms are turbo codes and related concatenated coding techniques [1]. Decoders for these codes consist of several concatenated soft-output decoders each of which decodes part of the overall code and then passes 'soft' reliability information to the other component decoders. The component soft-output algorithm prescribed in the original turbo code paper [1], usually known as the MAP or BCJR algorithm, is what we call the forward-backward algorithm [2][3]. This algorithm, originally proposed in the late 1960's, was generally overlooked in favor of the less complex Viterbi algorithm [4] [5] since an application that took advantage of soft information was not evident. In this paper, we describe techniques for implementing the forward-backward algorithm (FB) that are suitable for VLSI implementation.

The main idea in this paper can be summarized as extending well-known techniques used in implementing the Viterbi algorithm to the forward-backward algorithm. The FB algorithm can be thought of as two Viterbi-like algorithms running in opposite directions over the data, albeit with a slightly different computational kernel.

II. The Forward-Backward Algorithm

A. Background

It can be useful for a decoding algorithm to provide an estimate of the reliability of the decoded bits. The '*soft*' reliability values can be used to adjust the decoding algorithm to provide better performance. As far back as 1973, Forney proposed the use of 'augmented outputs' from the Viterbi algorithm as a measure of reliability of the decoding process [5]. Forney's heuristic idea to use the difference in state metrics between the best path and the next shortest path led to the soft-output Viterbi algorithm (SOVA) described by Batail [6] and Hagenauer and Hoeher [7].

It is important to define what is meant by soft information. The reliability of a decoded bit is best described by the a-posteriori probability (APP) $P(u|y)$. For an estimate of bit u (-1/+1) having received symbol y we define the *optimum soft output* as:

$$L(u) = \ln\frac{P(u=+1|y)}{P(u=-1|y)} \tag{1}$$

which is called the *Log-Likelihood Ratio* (LLR). The LLR is a convenient measure since it encapsulates both soft and hard bit information in one number. The sign of the number corresponds to the hard decision while the magnitude gives a reliability estimate.

The LLR can be easily computed by noticing that a decoding algorithm that uses the maximum a-posteriori (MAP) rule inherently calculates the APPs required. MAP algorithms have been proposed by several authors [2][3][8] but were generally ignored because the Viterbi algorithm can provide nearly identical hard outputs with less computational effort.

In this paper, we will focus on the practical implementation of what we call the “forward-backward algorithm” of [2][3]. In the literature, this algorithm is often referred to as the MAP algorithm or the BCJR algorithm, which are both somewhat inaccurate names. We propose the name “forward-backward” since it illustrates the progression of computation in the algorithm as described below. Abend and Fritchman proposed a variant of the MAP decoder that is optimal under the constraint of fixed decoding delay [8]. A simplified suboptimal detector (SPS) based on the Abend and Fritchman algorithm was derived in [9]. The Abend and Fritchman type algorithm has the drawback that the complexity of the decoder is exponentially related to the fixed decision delay. In [10] a MAP detector is derived that has linear dependence on the delay.

B. Description of the Algorithm

This description of the algorithm is based on [3] and [11] to which the reader is referred to for a detailed derivation. The algorithm is based on the same trellis as the Viterbi algorithm. The algorithm is performed on a block of N received symbols which corresponds to a trellis with a finite number of stages N. We will choose the transmitted bit u_k from the set of {-1,+1}. Upon receiving the symbol $\mathbf{y}_k$ from the AWGN channel with noise variance σ^2 we calculate the *branch probability* of the transition from state s' to state s as

$$\gamma_k(s', s) = \exp\left(\frac{-1}{2\sigma^2}\|\mathbf{y}_k - \mathbf{c}_k(s', s)\|^2\right) \qquad (2)$$

where $\mathbf{c}_k(s', s)$ is the expected symbol along the branch from state s' to state s.

The algorithm consists of three main steps:

Forward Recursion. The *forward state probability* of being in each state of the trellis at each time k given the knowledge of all the previous received symbols is recursively calculated and stored:

$$\alpha_k(s) = \sum_{s'}(\alpha_{k-1}(s')\gamma_k(s', s)) \qquad k = 1 \ldots N-1 \tag{3}$$

The recursion is initialized by forcing the starting state to state 0 and setting

$$\begin{aligned} \alpha_0(0) &= 1 \\ \alpha_0(s) &= 0 \qquad s \neq 0 \end{aligned} \tag{4}$$

Backward Recursion. The b*ackward state probability* of being in each state of the trellis at each time k given the knowledge of all the future received symbols is recursively calculated and stored:

$$\beta_{k-1}(s') = \sum_{s}(\beta_k(s)\gamma_k(s', s)) \qquad k = N \ldots 2 \tag{5}$$

The recursion is initialized by forcing the ending state to state 0 and setting

$$\begin{aligned} \beta_N(0) &= 1 \\ \beta_N(s) &= 0 \qquad s \neq 0 \end{aligned} \tag{6}$$

The trellis termination condition requires the entire block to be received before the backward recursion can begin.

Log-Likelihood ratio Calculation. The output LLR for each symbol at time k is calculated as

$$L(u_k) = \ln \frac{P(u_k = +1 | \mathbf{y}_k)}{P(u_k = -1 | \mathbf{y}_k)} \tag{7}$$

$$= \ln \frac{\sum_{\substack{(s', s) \\ u_k = +1}} \alpha_{k-1}(s')\gamma_k(s', s)\beta_k(s)}{\sum_{\substack{(s', s) \\ u_k = -1}} \alpha_{k-1}(s')\gamma_k(s', s)\beta_k(s)} \tag{8}$$

where the upper summation is over all branches with input label +1 and the lower summation is over all branches with input label -1.

We can simplify Equation 8 in the case where the two-state butterflies of the code are in the form of Figure 1. Recursive convolutional codes, such as those used in turbo codes [1] do not permit this simplification. The 'standard' non-recursive convolutional codes, however can be decoded much more easily using the following simplification.

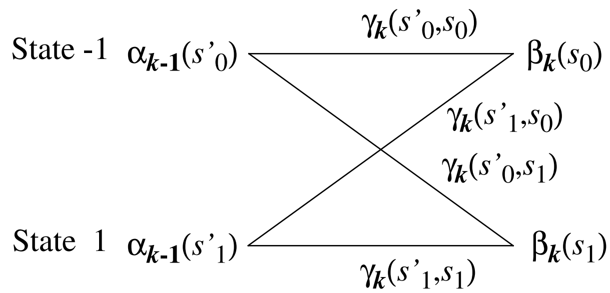

Figure 1: 2-state trellis.

The LLR calculation, for the two-state trellis gives:

$$L(u_k) = \ln\left(\frac{\alpha_{k-1}(s'_0)\cdot\gamma_k(s'_0,s_0)\cdot\beta_k(s_0)+\alpha_{k-1}(s'_1)\cdot\gamma_k(s'_1,s_0)\cdot\beta_k(s_0)}{\alpha_{k-1}(s'_0)\cdot\gamma_k(s'_0,s_1)\cdot\beta_k(s_1)+\alpha_{k-1}(s'_1)\cdot\gamma_k(s'_1,s_1)\cdot\beta_k(s_1)}\right) \qquad (9)$$

Then, by factoring the β terms:

$$L(u_k) = \ln\left(\frac{\beta_k(s_0)\cdot[\alpha_{k-1}(s'_0)\cdot\gamma_k(s'_0,s_0)+\alpha_{k-1}(s'_1)\cdot\gamma_k(s'_1,s_0)]}{\beta_k(s_1)\cdot[\alpha_{k-1}(s'_1)\cdot\gamma_k(s'_0,s_1)+\alpha_{k-1}(s'_1)\cdot\gamma_k(s'_1,s_1)]}\right) \qquad (10)$$

The terms in brackets are simply terms of the forward recursion (see Equation 3), thus Equation 9 can be rewritten:

$$L(u_k) = \ln\left(\frac{\beta_k(s_0)\cdot\alpha_k(s'_0)}{\beta_k(s_1)\cdot\alpha_k(s'_1)}\right) \qquad (11)$$

This simplification of the $L(u_k)$ equation can be generalized for a 2^{ν} state trellis. In fact, a 2^{ν} state trellis is composed of $2^{\nu-1}$ 2-state trellises, where the i$^{\text{th}}$ (i = b$i_{\nu\text{-}1}i_{\nu\text{-}2}...i_0$ in binary) two-state trellis connects nodes s'_0 = b$i_{\nu\text{-}1}i_{\nu\text{-}2}...i_0$0 and s'_1 = b$i_{\nu\text{-}1}i_{\nu\text{-}2}...i_0$1 with nodes s_0 = b0$i_{\nu\text{-}1}i_{\nu\text{-}2}...i_0$ and s_1 = b1$i_{\nu\text{-}1}i_{\nu\text{-}2}...i_0$. Then, starting from Equation 8 and performing the same computation from Equation 8 to Equation 10 for each of the two-state trellises, the new expression for $L(u_k)$ becomes:

$$L(u_k) = \ln\left(\frac{\sum_{0\le i<2^{\nu-1}}\alpha_k(s_i)\cdot\beta_k(s_i)}{\sum_{2^{\nu-1}\le j<2^{\nu}}\alpha_k(s_j)\cdot\beta_k(s_j)}\right) \qquad (12)$$

This expression is easier to compute than the previous version.

The forward-backward algorithm as described requires the entire message to be received before decoding can start. For real-time applications, this restriction can be too stringent. Similar to the Viterbi algorithm, we can start the backwards recursion from the “all-zero vector” β^0 with data $\{y_k\}$, k from n down to n-L. L iterations of backwards recursion allows us to reach a very good approximation of $g.\beta_{n-L}$ (where g is a positive multiplicative factor). This multiplicative coefficient doesn’t affect the value of the LLR. In the following, we will consider that after L cycles of backward recursion, the resulting state metric vector is the correct one. This property can be used in a hardware realization to start the effective decoding of the bits before the end of the message.

L is called the convergence length. Its typical value is a few times the constraint length of the encoder (typically between 5 to 10 times), increasing with decreasing SNR.

C. The Forward-Backward Algorithm in the Logarithmic Domain

The forward-backward algorithm was virtually ignored for many years because of the difficulty in implementing efficient exponentiation and multiplication. If the algorithm is implemented in the logarithmic domain like the Viterbi algorithm, then the multiplications become additions and the exponentials disappear. Addition is transformed according to the rule described in [12]. Following Erfanian and Pasupathy, who first applied this rule to the Abend and Fritchman MAP algorithm [9], the additions are replaced using the Jacobi logarithm:

$$\begin{aligned} MAX^*(x, y) &= \ln(e^x + e^y) \\ &= MAX(x, y) + \ln(1 + e^{-|x-y|}) \end{aligned} \tag{13}$$

which is called the MAX^* operation to denote that it is essentially a maximum operator adjusted by a correction factor. The second term, a function of the single variable x - y, can be precalculated and stored in a small lookup table with negligible effects on performance [9].

The forward-backward algorithm will now be restated in the logarithmic domain. As with the Viterbi algorithm, logarithms of probabilities are referred to as *metrics*. Define the new quantities:

Branch Metrics:

$$G_k(s', s) = \ln(\gamma_k(s', s)) \tag{14}$$

Forward State Metrics:

$$A_k(s) = \ln(\alpha_k(s)) \tag{15}$$

Backward State Metrics:

$$B_k(s) = \ln(\beta_k(s)) \tag{16}$$

The branch metric calculation eliminates the exponential:

$$G_k(s', s) = \frac{-1}{2\sigma^2}\|\mathbf{y}_k - \mathbf{c}_k(s', s)\|^2 \tag{17}$$

The forward state metric recursion becomes:

$$A_k(s) = \underset{s'}{MAX^*}(A_{k-1}(s') + G_k(s', s)) \qquad k = 1 \ldots N-1 \tag{18}$$

with initial conditions:

$$\begin{aligned} A_0(0) &= 0 \\ A_0(s) &= -\infty \qquad s \neq 0 \end{aligned} \tag{19}$$

The backward state metric recursion becomes:

$$B_{k-1}(s') = \underset{s}{MAX^*}(B_k(s) + G_k(s', s)) \qquad k = N \ldots 2 \tag{20}$$

with initial conditions:

$$\begin{aligned} B_N(0) &= 0 \\ B_N(s) &= -\infty \qquad s \neq 0 \end{aligned} \tag{21}$$

The dynamic range of the metrics is much lower than the associated probabilities.

The computational kernel of the algorithm is analogous to the Add-Compare-Select (ACS) operation in the Viterbi algorithm adjusted by an offset as a correction factor. This kernel will be named ACSO (Add-Compare-Select-Offset) in the following.

The log-likelihood ratio calculation becomes

$$L(u_k) = \underset{\substack{(s', s) \\ u_k = +1}}{MAX^*}(A_{k-1}(s') + G_k(s', s) + B_k(s)) - \underset{\substack{(s', s) \\ u_k = -1}}{MAX^*}(A_{k-1}(s') + G_k(s', s) + B_k(s)) \tag{22}$$

The simplification of Equation 12 can also be applied in the log domain. The LLR expression implies the logarithm of a multi-term sum of products. Since addition is a commutative operator, this property should still exist in the log domain with the MAX^* operator. In fact:

$$\begin{aligned}\log(e^a + e^b + e^c + e^d) &= \log(e^{\log(e^a + e^b)} + e^{\log(e^c + e^d)}) \\ &= MAX^*(MAX^*(a, b), MAX^*(c, d))\end{aligned} \tag{23}$$

Thus, we can write:

$$L(u_k) = \underset{0 \le i < 2^{\nu - 1}}{MAX^*} (A_k(s_i) + B_k(s_i)) - \underset{2^{\nu - 1} \le j < 2^{\nu}}{MAX^*} (A_k(s_i) + B_k(s_i)) \tag{24}$$

D. Upper Bounds for MAX^*

All the following upper bounds are derived from the definition of MAX^* in eq. (13).

$$MAX^*(x, y) \ge MAX(x, y) \ge (x, y) \tag{25}$$

For practical implementation, one can notice that, due to the finite precision of the hardware implementation, the function $\ln(1 + e^{-|x-y|})$ gives a zero result as soon as $|x\text{-}y|$ is large enough. For example, if the values are coded in fixed precision with 3 digits after the dot (quantum of 0.125), then $|x\text{-}y| > 2.5 \Rightarrow \ln(1 + e^{-|x-y|}) < 0.079$, thus it will be rounded to 0. In that case, the computation of the offset of the MAX^* operator can be performed with two pieces of information: a boolean Z (for zero) that indicates if $|x\text{-}y|$ is above or equal to the first power of two greater than 2.5, i.e. 4. If Z is true, then the offset is equal to 0. If not, its exact value is computed with the 5 least significant bits of $|x\text{-}y|$. A look-up-table is the most straightforward way to perform this operation. In the general case, there is a positive value δ so that:

$$MAX^*(x, y) = MAX(x, y) \qquad if \qquad |x - y| > 2^{\delta} \tag{26}$$

We also have

$$MAX^*(x, y) \le MAX(x, y) + \ln(2) \tag{27}$$

With equality if $x = y$.

The following sections deal with VLSI implementations of the FB algorithm. Section 3 studies the problem of internal representation of the state metrics for a fixed point implementation. Section 4 focuses on efficient architectures to realize a forward (or backward) recursion. Section 5 describes techniques for implementing the LLR calculation. Finally, Section 6 proposes several schedules for the forward and backward recursions. As the problem of the computation of the forward and the backward recursion is symmetrical (see Equation 18 and Equation 20) in time (i.e. identical in terms of hardware computation) only the forward recursion is described in Sections 3 and 4.

III. Precision of the State Metrics

The precision of the state metrics is an important issue for VLSI implementation. The number of bits n_{sm} used to code the state metrics determines both the hardware complexity and the speed of the hardware. This motivates the need for techniques which minimize the number of bits required without modifying the behavior of the algorithm.

The same problem has been intensively studied for the Viterbi algorithm and solutions using rescaling or modulo 2^n arithmetic are widely used [13][14]. These techniques are based on the fact that, at every instant k, the dynamic range $\Delta(k)$ of a state metric (i.e. the difference between the state metrics with the highest and lowest values), is bounded by Δ_{max}.

The forward recursion in the FB algorithm is slightly different than the Viterbi algorithm since:

(i) The outputs of the recursion are the state metrics themselves and not the decisions of the ACS.

(ii) The Add-MAX* is an ACS operation with an added correction factor.

These differences lead us to question if the well-known implementation techniques for the Viterbi algorithm are also applicable to the forward-backward algorithm. The first part of this section shows that the LLR result is independent of a global shift of all of the state metrics of the forward and backward recursions. Afterwards, bounds on the dynamic range of the state metric are given.

A. Rescaling the State Metrics.

Let us first show that the MAX^* operator is shift invariant, that is it still produces a valid result if both of its arguments have a common constant added to them. Let x, y and z be real numbers. From the definition of MAX^*, it follows that:

$$MAX^*(x+z, y+z) = MAX(x+z, y+z) + \ln(1 + e^{-|(x+z)-(y+z)|}) \qquad (28)$$

Thus

$$MAX^*(x+z, y+z) = MAX(x, y) + z + \ln(1 + e^{-|x-y|}) = MAX^*(x, y) + z \qquad (29)$$

According to Equation 29, the MAX^* operation is linear. Thus, a global shift of Z for all $A_{k-1}(s')$ values (or $B_k(s)$) in Equation 18 won't change the value of $L(u_k)$ since the contribution of Z, when put outside the two MAX^* operators, is cancelled. Thus, it is the difference between the state metrics and not their absolute values that are important: rescaling of the state metrics can be performed.

B. An Approximate Bound on the Dynamic Range of the State Metrics.

Let us define l as the minimum number so that, for all $k>0$, there is a path through the trellis between every state at time k and $k + l$. Let us define M as the maximum absolute value of the branch metric. Then, for all $k \le l$, a rough bound on the dynamic range of the state metrics is:

$$\Delta(k) \le (2 \cdot M + \ln(2)) \cdot l \qquad (30)$$

Proof: Let $sm_{max}(k)$ and $sm_{min}(k)$ be respectively the maximum and minimum value of the state metric at time k. Then, according to the definition of l, there is a path of length l in the trellis between every state s at time k and the state s_m with value $sm_{max}(k-l)$ at time k-l. Since at every step, the maximum state metric (with, eventually, a positive correction factor) is taken, in the worst case, among the path between s_m and s, the state metric can decrease by M at each step. Thus, $sm_{min}(k)$ is at least equal to or greater than $sm_{max}(k-l)$ - Ml. Similarly, the maximum increase at each stage of the state metric among this path is $M + \ln(2)$ ($\ln(2)$ is the maximum value of the correction factor added at each stage), thus, $sm_{max}(k)$ is lower than, or equal to $sm_{max}(k-l) + Ml + \ln(2)l$. Grouping the upper bound of $sm_{min}(k)$ and lower bound of $sm_{max}(k)$ leads to Equation 30.

Note: in case of a trellis corresponding to a shift register of length ν, l is equal to ν.

C. A Finer Bound on the Dynamic Range of the State Metrics for a Convolutional Decoder.

A more precise bound can be obtained in the case of a convolutional decoder based on the intrinsic properties of the encoder. All the developments are based on previous work for the Viterbi algorithm [15].

- **Simplification of the computation of the branch metrics for a convolutional decoder**

In the case of a rate 1/r convolutional code, $c_k(s',s)$ is a r-dimensional vector with elements {-1, +1} (or {-1, 0, +1} in the case of a punctured code). Using Equation 29, the computation of the branch metrics (Equation 17) can be extended and simplified.

$$G_k(s', s) = \frac{-1}{2\sigma^2}(\|\mathbf{y}_k\|^2 + \|\mathbf{c}_k(s', s)\|^2) + \frac{1}{\sigma^2}(\mathbf{y}_k \cdot \mathbf{c}_k(s', s)) \tag{31}$$

The first terms are common to all branch metrics, thus they can be dropped. The last terms can be decomposed on the r dimensions of the vector. Thus, the modified branch metrics $G'_k(s',s)$ are:

$$G_k(s', s) = \frac{1}{\sigma^2} \cdot \sum_{0 \le i < r} y_k^i \cdot \mathbf{c}_k(s', s)^i \qquad \mathbf{c}_k(s', s)^i \in \{-1, 0, 1\} \tag{32}$$

This expression can be used to find a more precise bound of $\Delta(k)$.

- **Precise bound on $\Delta(k)$**

In the case of a Viterbi decoder, methods leading to the exact bound of the state metric differences have been described [15]. The method described in [15] can be adapted to the MAX* recursion. It gives only a lower bound Δ' of the bound Δ since its defines a construction method to obtain an input sequence leading effectively to $\Delta(k) = \Delta'$. Although not demonstrated formally, the obtained bound Δ' seems to be equal to Δ.

The principle is to receive a sequence {S} of symbols corresponding to the maximum reliability (i.e. maximum values of the branch metric, according to Equation 32) of the “all-zero” path. Then starting with all state metrics at zero at time k=0, the series {S} is decoded. At each iteration, $\Delta(k)$ is computed. The value Δ' is the limit of $\Delta(k)$ when k goes to infinity.

Note: as the encoder is linear, the value of Δ' doesn’t depend on the path on the trellis used to find Δ'.

As an example, let us consider a recursive systematic encoder with generator polynomials (7,5) (Figure 2). Moreover let us assume that the received symbols $\{y_k^i\}$ are saturated between -2 and 2. Table 1 shows the evolution of the state metrics if at every time k, y_k = (-2,-2) (maximum reliability for branch between state 0 to state 0.

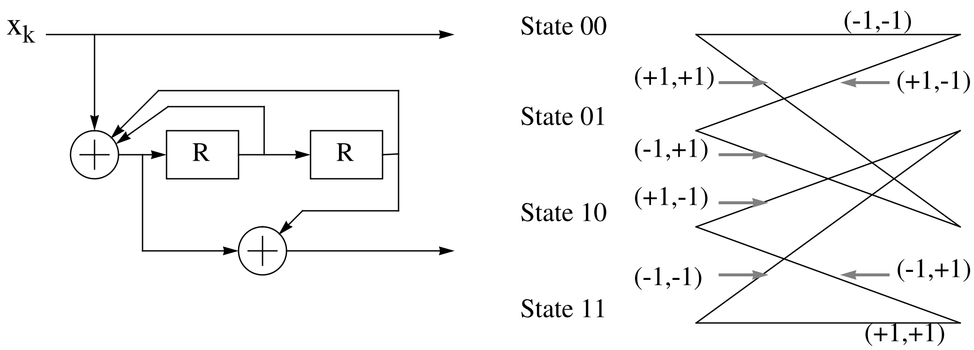

Figure 2: State transitions of the recursive systematic encoder with polynomials (7,5).

	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=13	t=20
0	0.000	4.000	8.000	12.000	16.000	20.000	24.000	28.000	52.000	80.000
1	0.000	0.693	4.036	5.117	8.105	9.642	12.791	16.127	40.038	68.038
2	0.000	4.000	4.702	8.054	9.400	12.747	16.091	20.040	44.019	72.019
3	0.000	0.693	4.702	5.117	8.105	9.642	12.791	16.127	40.038	68.038
$\Delta(k)$	0.000	3.307	3.964	6.883	7.895	10.358	11.209	11.873	11.963	11.963

Table 1: Variation of the state metrics after t stages on the all-zero path.

As shown in Table 1, after 13 iterations, the value of $\Delta(k)$ doesn't increase any more. The limit value Δ' = 11.963 is the maximum value of the branch metric dynamic. In the case of fixed point computation, the same method can be applied.

The approximate bound of Equation 30 is, in this example, 17.386. The bound obtained by the proposed method is more precise and can lead to more efficient hardware realization.

IV. Architecture for the Forward and Backward Recursions

This section is divided into three parts. The first part describes the architecture to compute the forward state metrics. Then, we make an original proposition of the position of the register for the recursion loop in order to increase the speed of the architecture. Finally, the last section proposes a MSB first computation of the forward recursion in order to multiplex two recursions on the same hardware.

A. Computation of the Forward State Metrics: the ACSO unit

According to Equation 18 the architecture of the processing unit that computes a new value of $A_k(s)$ is shown in Figure 3. The structure consists of the well known ACS (Add Compare Select) unit used for the Viterbi algorithm (grey area in Figure 3) and some extra hardware to generate the "offset" corresponding to the correction factor $\ln(1 + e^{-|x-y|})$ of Equation 13.

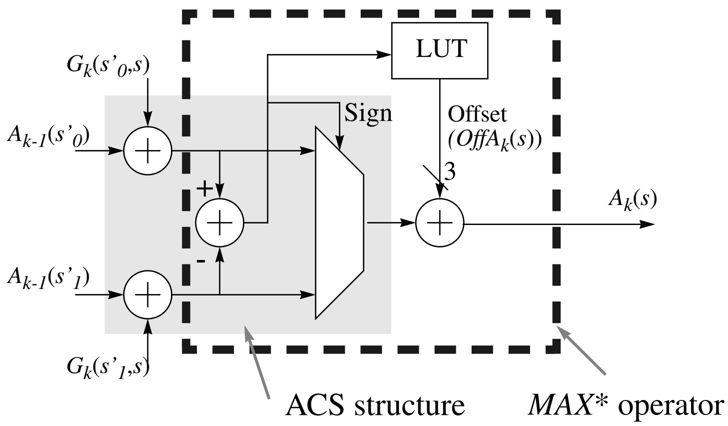

Figure 3: Architecture of an ACSO.

The offset is generated directly with a look-up-table (LUT) that contains the pre-calculated result of $\ln(1 + e^{-|x-y|})$. Then, the offset is added to the result of the ACS operation to generate the final value of $A_k(s)$. In the following, we will call this processor unit an Add Compare Select Offset (ACSO) unit.

B. Architecture for the Forward State Metric Recursion

The natural way to perform the forward state metric recursion is to place a register at the output of the ACSO unit, in order to keep $A_k(s)$ for the next iteration. This architecture is the same as the one used for the Viterbi algorithm and all the literature on the speed-area trade-offs for the ACS recursion can be reused for the ACSO computation. Nevertheless, there is another position for the register that can reduce the critical path of the recursion loop. Figure 4 shows two steps of a two-state trellis.

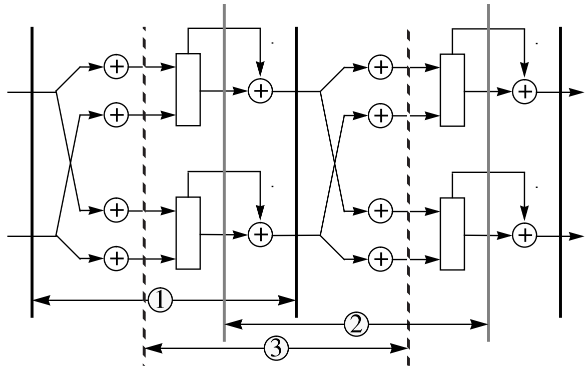

Figure 4: Three different positions of the register in the data flow of the forward (or backward) algorithm leading to 3 types of ACSO recursion architectures.

Three different positions of the recursion loop register are shown. The first position is the classical one. It leads to an ACSO unit. The second position leads to an CSOA unit while the third position leads to an OACS unit. The last one, the OSCA unit shown in Figure 5, has a smaller critical path compared to the ACSO unit. Briefly, in the case of a ACSO unit, the critical path is composed of the propagation of the carry (t_c) in the first adder, the propagation of one full-adder (t_{fa}) for the comparison (as soon as a result of the sum is available, it can be used for the comparison), the time of the LUT access (t_{LUT}) and the multiplexer (t_{mux}), and then, once more, the time of the propagation of the carry in the offset addition. For the OACS unit, the critical path is only composed of the propagation of the carry in the first adder (the addition of the offset), the propagation of one full-adder for the addition of the branch metric, another propagation of one full-adder for the comparison, and then, the maximum of the LUT access and the multiplexer. Thus, the critical path is decreased from

$$t_{acso} = n_{sm} \cdot t_c + t_{fa} + max(t_{LUT}, t_{mux}) + n_{sm} \cdot t_c \tag{33}$$

to

$$t_{oacs} = n_{sm} \cdot t_c + 2 \cdot t_{fa} + max(t_{LUT}, t_{mux}) \tag{34}$$

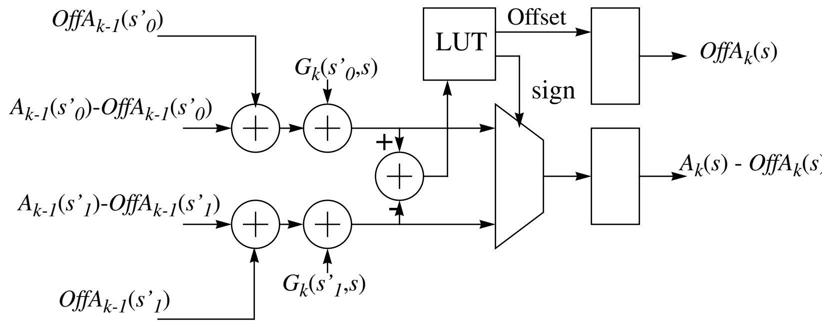

Figure 5: Architecture of an OACS unit.

The decrease of the critical path is paid for by the additional register needed to store the offset value between two iterations. The area-speed trade-off is determined by the specification of the application.

Remark: The critical path of the OACS unit can be further decreased by using a dual LSB-MSB comparison. The idea is based on the following remarks:

(i) $\mathrm{Max}^*(A,B) = \mathrm{Max}(A,B)$ if $|A-B| > 2^t$ (from Equation 26).

(ii) the number of bits n_{bm} used to code the branch metric is lower than n_{sm}.

(iii) the number of bits n_{offset} used to code the offset is lower than n_{sm}.

Thus, if the first MSB of the two input words differ, whatever the offset and the branch metric are, it can be predicted with certainty that the next offset $Off(A_k(s))$ will be null and that $A_k(s)$ will be equal to the maximum of $A_{k-1}(s'_0)$ and $A_{k-1}(s'_1)$. These properties can be used to further reduce the critical path of the OACS loop.

V. LLR Architecture

The last step of the FB algorithm is the computation of the LLR value of the decoded bit. Parallel architectures for the LLR computation can be derived directly from Equation 22 or Equation 24. The first stage is composed of 2^ν adders. The second stage is composed of two $2^{\nu-1}$ operand MAX* operators. Finally, the last operation is the subtraction. A tree architecture can be used for the hardware realization of the $2^{\nu-1}$-operand MAX^* operators.

An efficient hardware architecture to perform the summation of 2^a terms is to use a binary tree of adders. With this architecture, there are *a* layers of adders between the input and the result and the total number of adders is 2^a-1. The same structure can be used to perform the $2^{\nu-1}$-operand MAX^*. The two-operand MAX^* operator is shown in the dashed rectangle in Figure 3. The whole critical path t_{TMAX^*} is (ν-1) times the critical path of each layer (i.e. *MAX** operator), thus:

$$t_{TMAX^*} = (\nu - 1) \cdot (t_{FA} + max(t_{LUT}, t_{mux}) + n_{sm} \cdot t_c) \tag{35}$$

In fact, the first adder can start its computation as soon as the partial result of the previous sum is available (computation of the sum for the first layer, the addition of the offset of the previous layer for the other layer).

VI. General Architecture

All the elements of the FB architecture have now been described. The last part of our survey on VLSI architectures for the FB algorithm is the overall organization of the computation. Briefly speaking, the generation of the LLR values requires both A_k and B_k values, which are generated in chronologically reverse order. The first implication is that, somehow, memory is needed to store a given type of vector (say, Ak), until the corresponding vector (B_k) is generated. Each state metric vector is composed of 2^ν state metrics (the size of the trellis), each one n_{sm} bits wide. The total number of bits for each vector is large (2^ν n_{sm}) and thus, reducing the number of state metrics to store is an important issue in minimizing the implementation area.

The first part of this section describes the architecture of a high speed VLSI circuit for the forward algorithm. Then, though different steps, we propose several organizations of computation that reduce the number of vectors that need to be stored by up to a factor of eight.

A. Existing Solution

The first real-time FB VLSI architectures in the literature are described in [16][17]. The architecture of [17] is based on three recursion units (RU), two used for the backward recursion (RU_{B1} and RU_{B2}), and one forward Unit (RU_A). Each RU contains 2^ν MAX^* operators working in parallel so that one recursion can be performed in one clock cycle. The two backward RUs play a role similar to the two trace-back units in the Viterbi decoder of [18]. Let us use the same graphical representation as in [17][19][20] to explain the organization of the computation. In Figure 6, the horizontal axis represents time, with units of a symbol period. The vertical axis represents the received symbol. Thus, the curve (x = y) shows that, at time $t = k$, the symbol $\{y_k\}$ becomes available. Let us describe how the L symbols $\{y_k\}_{L \le k < 2L}$ are decoded (segment I of Figure 6). From $t = 3L$ to $4L$-1, RU_{B1} performs L recursions, starting from y_{3L-1} down to y_{2L} (segment II of Figure 6). This process is initialized with the "all-zero" state vector B^0, but after L iterations, as noted in [17], convergence is reached and B_{2L} is then obtained. During those same L cycles, RU_A generates the vectors $\{A_k\}_{L \le k < 2L}$ (segment III of Figure 6). The vectors $\{A_k\}_{L \le k < 2L}$ are stored until they are needed for the LLR computation (grey area of Figure 6). Then, between $t = 3L$ and $4L$-1, RU_{B2} starts from state B_{2L} to compute B_{2L-1} down to B_L (segment IV of Figure 6). At each cycle, the vector A_k corresponding to the B_k computed is extracted from the memory in order to compute $L(u_k)$. Finally, between $t = 5L$ and $6L$-1, the data are reordered (segment V Figure 6). The same process is then re-iterated every L cycles, as shown in Figure 6. We note that if the FB unit is being used in a turbo decoder, the reordering can be done implicitly by the interleaver. However, the a-priori information to be subtracted [1] might still have to be reordered.

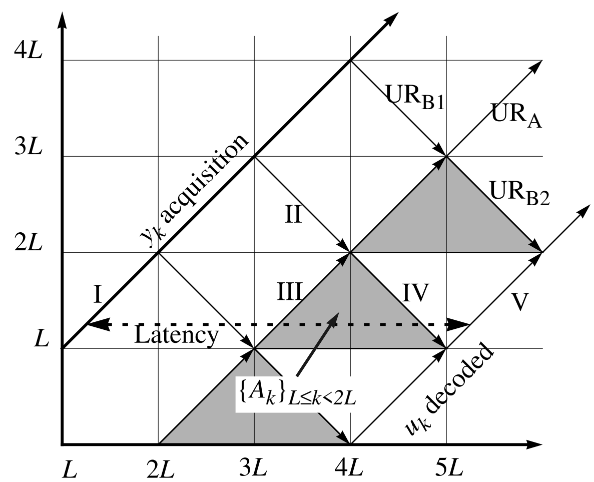

Figure 6: Graphical representation of a real time FB architecture.

This graphical representation gives some useful information about the architecture. For example, the values of:

(i) the decoding latency: D = $4L$ (horizontal distance between the array "acquisition" and "decoded bit").

(ii) the number of A vectors to be stored: M = L (maximum vertical size of the grey area).

(iii) the "computational cost" of the architecture, i.e the total number of forward and backward iterations performed for each received data: P = 3 (the number of arrows of RU cut by a vertical line).

In what follows, this architecture is referred to as (n_A = 1, n_B = 2, M_A), where n_A and n_B are respectively the number of RUs used for the forward and backward recursions respectively. M_A (for the memory of state metric *A*) indicates that the *A* vectors are stored for the LLR bit computation. Note that in this architecture, the forward recursion is performed 2*L* cycles after the initial reception of data.

With the (n_A= 1, n_B = 2, M_A) architecture, *L* state vectors have to be stored. The length of convergence *L* is relatively small (a few times the constraint length ν) but the size of the state vector is very large. In fact, a state vector is composed of the 2^ν states metrics, each state metric is n_{sm} bits wide, i.e. $2^\nu n_{sm}$ bits per state metric vector. The resulting memory is very narrow and thus, not well suited for a realization with RAM blocks.

Other architectures have been developed in the paper. Each present different trade-offs between computational power, memory size and memory bandwidth. Their graphical representations are given below.

B. (n_B= 1, n_A = 2, M_B) Architecture

This architecture is equivalent to the former one, except that the forward recursion is performed 4*L* cycles after the reception of the data, instead of 2*L* cycles (see Figure 7). In this scheme, the *B* vectors generated by RU_{B2} are stored until the computation of the corresponding *A* vectors by RU_A. Then, the LLR values are computed in the natural order.

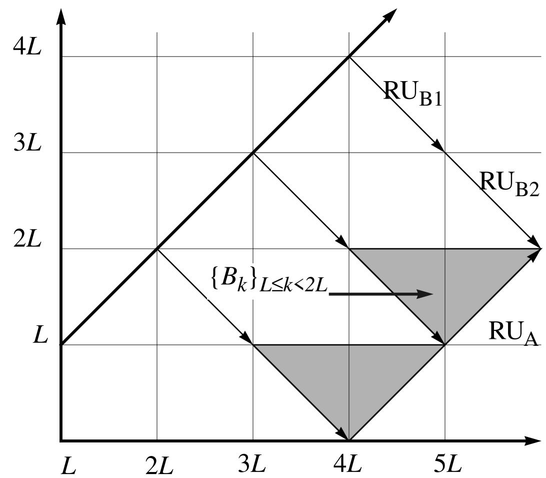

Figure 7: Graphical representation of the (n_A=1, n_B = 2, M_B) architecture.

The complexity of this architecture is equivalent to the (n_A= 1, n_B = 2, M_A) one, except that the reordering of the results is not required. Moreover, in the context of a turbo decoder circuit, there is an additional advantage since the extrinsic information can be directly subtracted from the decoded bit.

C. (n_A= 1, n_B = 2, $M_{(A+B)/2}$) Architecture

In this architecture, the forward recursion is performed 3*L* cycles after the reception of the data (see Figure 8). Thus, *L*/2 *A* vectors and *L*/2 *B* vectors have to be stored. The total number of state vectors to be stored is still *L*. Moreover, with this solution, *L* bits have to be decoded in the last *L*/2 clocks cycles of an iteration, thus, two APP units have to be used. This scheme becomes valuable in the context of a turbo-decoder. Since two FB algorithms are performed in parallel, it is possible to share the *L* memory words between the two FB algorithms by an appropriate interleaving of the two operations, as shown in Figure 8. In this figure, the second iteration is represented with dotted lines and the corresponding vector memory with a striped region.

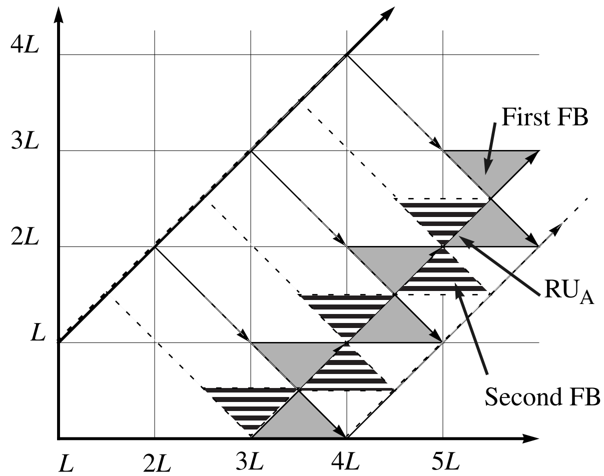

Figure 8: Graphical representation of the (n_A=1, n_B = 2, $M_{(A+B)/2}$) architecture.

With this interleaving, the amount of memory for state metrics corresponding to each FB is divided by two. Thus, the final area will be smaller than the simple juxtaposition of two “classical” FB algorithms.

The FB architecture can use more than two RUs for the backward recursion and/or more than one RU for the forward recursion. The following sections describe some interesting solutions.

D. (n_A=1, n_B = 3, M_B) Architecture

An additional backward unit leads to the schedule of Figure 9.

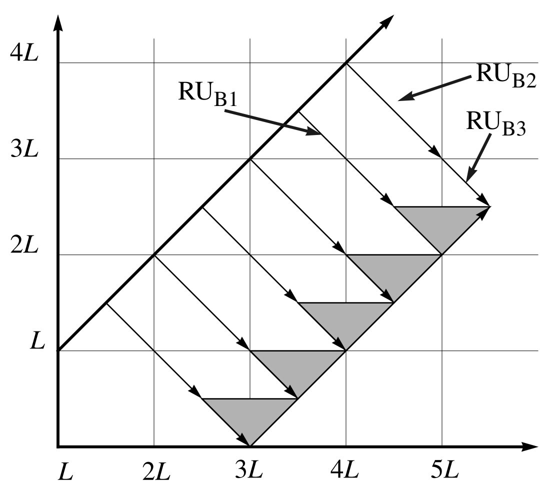

Figure 9: Graphical representation of the (n_A=1, n_B = 3, M_B) architecture.

A new backward recursion is started every $L/2$ cycles on a length of $L+L/2$ symbols. The first L steps are used to achieve convergence, then the last $L/2$ steps generate $L/2$ vectors B. The new latency is now $3L$ and the amount of memory needed to store the B vectors is only $L/2$. Two observations are worth noting:

(i) the reduction of the latency and the memory size is paid for by a new backward unit.

(ii) a solution of type ($n_A = 2$, $n_B = 2$, $M_{(A+B)/2}$) can also be used.

E. (n_A=2, n_B=2, M_B) Architecture

The addition of an extra forward unit can also decrease the State Vector Memory (SVM) by a factor of two, as shown in Figure 10.

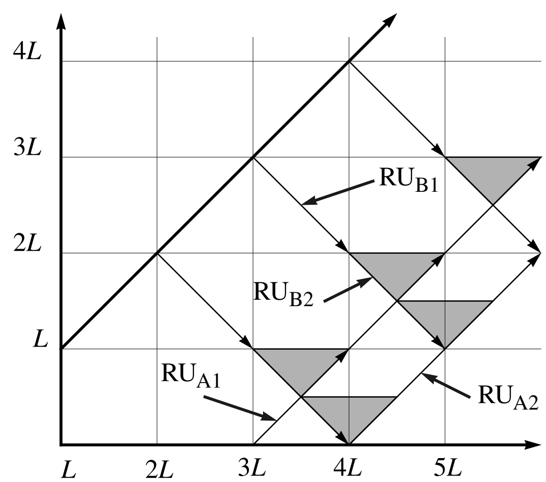

Figure 10: Graphical representation of the (n_A=2, n_B = 2, M_B) architecture.

This scheme has the same number of processing units ($n_A+n_B = 4$) and the same state metric memory size $M = L/2$ as the (n_A=1, n_B=3, M_B) architecture but its latency is $4L$ compared to $3L$ for the architecture of the previous section. However, the second recursion unit RU_{A2} can be simplified, since it only copies, with a time shift of L cycles, the computation of RU_{A1}. Thus, there exists a trade-off between computational complexity and memory. By storing, during L cycles, each decision and offset value generated by RU_{A1}, the complexity of RU_{A2} is almost divided by two (see Figure 11).

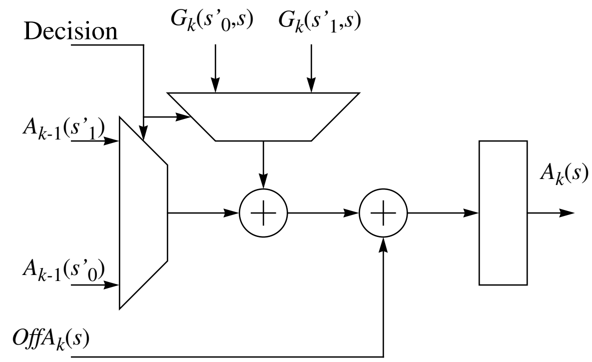

Figure 11: Simplified ACSO unit.

This method is very similar to the method used for the Soft Output Viterbi Algorithm in [7].

Note: Once more, a ($n_A = 2$, $n_B = 2$, $M_{(A+B)/2}$) method can also be used.

F. ($n_A = 2$, $n_B = 2$, M_A, Pt_B) Architecture

This new type of architecture is a generalization of the idea described above: instead of memorizing a large number of *A* (or *B*) vectors, they are re-computed when they are needed. For this, the context (i.e the state metrics) of an iteration process is saved in a pointer. This pointer is used later to recompute, with a delay, the series state metric. Such a process is given in Figure 12.

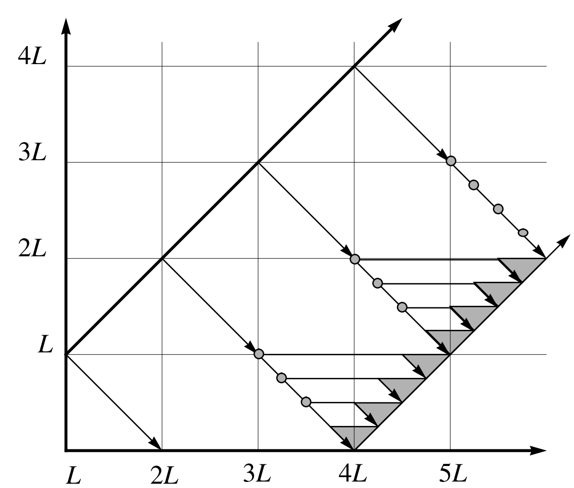

Figure 12: Graphical representation of the (n_A=1, n_B = 3, M_B, Pt_B) architecture.

In this scheme, the state metrics of RU_{B2} are saved every $L/4$ cycles (small circles in Figure 12). Those 4 state metrics are used as a seed, or pointer, to start the third backward process (RU_{B3}, in Figure 12) of length $L/4$. The third backward recursion is synchronized with the forward recursion in order to minimize the size of the B vector to be stored. In practice, only 3 seeds are needed since RU_{B2} and RU_{B3} process the same data during the last quarter of a segment of L cycles. With this method, the latency is still $4L$ but the number of state metrics to store is now $3 + L/4$. With such a small number of vectors, the use of a registers instead of RAM can be used to store the state metrics. This avoids the use of a RAM with an unusual aspect ratio with the consequent negative impact on performance.

This scheme becomes particularly attractive if two FB algorithms are implemented in a single chip (as it is the case for a turbo-decoder), since a (n_A=2, n_B=2, $M_{(A+B)/2}$,Pt_B) architecture can be used to share the 6 + L/4 vectors of the two FB algorithms (see Figure 13).

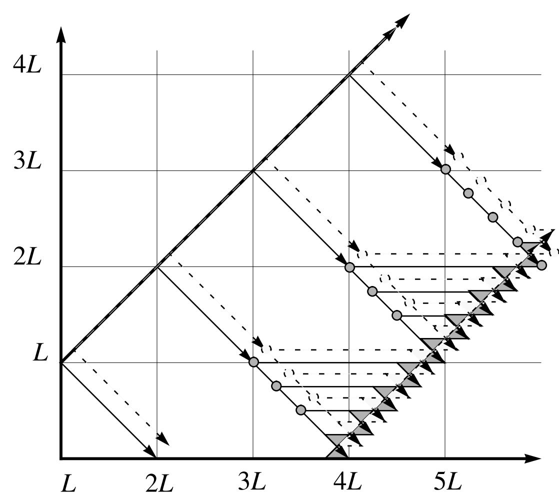

Figure 13: Graphical representation of the (n_A=1, n_B = 3, $M_{(A+B)/2}$, Pt_B) architecture.

This scheme is very efficient because it avoids the use of a RAM for storing the state metrics.

G. *Generalization of the Architecture*

Many combinations of the above architectures can be realized, each one with its own advantages and disadvantages. In the above examples, the ratio p between the hardware clock and the symbol clock was 1. More architectures can be uncovered by loosening this restriction. For example, if this ratio p is 2 (i.e. two clock cycles for each received symbol), the speed of the RU is doubled. Thus, an architecture like, for example, (p=2, n_A=1, n_B=1, M_A) can be used (see Figure 14).

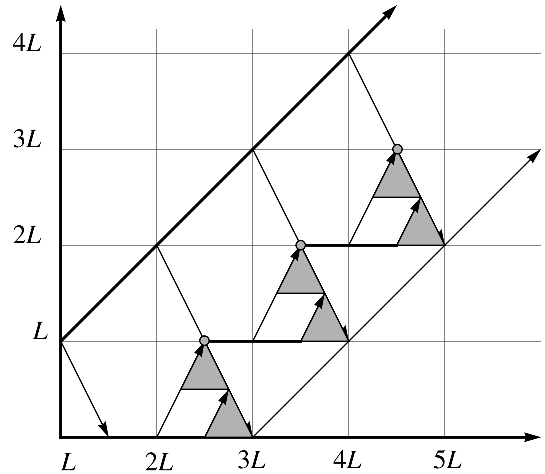

Figure 14: Graphical representation of the (p=2, n_A=1, n_B = 1, M_A, Pt_{1A}) architecture.

H. Summary of the Different Configurations

In Table 2, the different configurations are evaluated with different criteria in order to help the designer of a system.

n_A	n_B	**Memory organization**	**Recursion Units**	***Memory***	***Latency***	**notes**
1	2	M_A or M_B	3	L	4L	Figure 6, Figure 7
1	2	$M_{(A+B)/2}$	3	L/2	4L	Figure 8
1.5	2	M_B	3.5	L/2	4L	Figure 10
1	2.5	M_B, Pt_B	3.5	L/4+4	4L	Figure 12
1	2.5	M_A, Pt_B	3.5	L/p+p-1	4L	p = number of pointers
1	2.5	$M_{(A+B)/2}$, Pt_B	3.5	L/8+3	4L	Figure 13
1	3	M_B	4	L/2	3L	Figure 9
1+0.5n	2	M_A or M_B	3 + 0.5n	L/n	4L	n = number of half RU_B
1	2+n	M_A or M_B	3 + n	$L/(n+1)$	$2L+L/(n+1)$	n = number of RU_B

Table 2: Performance summary of the different architectures.

We can see that in the case of two FB algorithms implemented together in the same circuit, it is possible to decrease the number of vectors from L to L/8+3. This reduction allows the realization of this memory using only registers.

VII. Conclusion

We have presented a survey of techniques for VLSI implementations of the forward-backward algorithm. As a general conclusion, the well-known results from the Viterbi algorithm literature can be applied to the forward-backward algorithm. The computational kernel of the FB algorithm is very similar to that of the ACS of VA algorithm with an added offset. Some careful thinking reveals that it is better to add the offset first and then do the ACS operation in order to reduce the critical path of the circuit (OACS). A general architecture for the FB algorithm was developed which exposes some interesting trade-offs for VLSI implementations. Most importantly, we have exposed architectures which eliminate the need for RAMs with a narrow aspect ratio and possibly allow the RAM to be replaced altogether with registers. An architecture which shares a memory bank between two FB decoders enables efficient implementation of turbo decoders. An IC which incorporates some of the ideas in this paper is currently being designed.

VIII. References

[1] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," *Proc. ICC'93*, Geneva, Switzerland, May 23-26, 1993. pp. 1064-1070.

[2] R. W. Chang and J. C. Hancock, "On Receiver Structures for Channels Having Memory," *IEEE Transactions on Information Theory*, Vol. IT-12, No. 4. pp. 463-468. October 1966.

[3] 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*, pp.284-287. March 1974.

[4] A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Transactions on Information Theory*, Vol. 13, No. 2, pp. 260-269, April 1967.

[5] G. D. Forney Jr., "The Viterbi Algorithm," *Proceedings of the IEEE*, vol. 61, pp. 268-278, Mar. 1973.

[6] G. Battail, "Ponderation des symboles decodes par l'algorithme de Viterbi," *Ann.*

Telecommun., vol. 42, pp. 31-38. Jan. 1987.

[7] J. Hagenauer and P. Hoeher, "A Viterbi Algorithm with Soft-Decision Outputs and its Applications," *Proceedings IEEE Globecom Conference*, Dallas TX. pp. 1680-1686. Nov. 1989.

[8] K. Abend and B. Fritchman, "Statistical Detection for Communication Channels with Intersymbol Interference," *Proceedings of the IEEE*, Vol. 58, No. 5, pp. 779-785. May 1970.

[9] J. A. Erfanian and S. Pasupathy, "Low-Complexity Parallel-Structure Symbol-by-Symbol Detection for ISI Channels," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, June 1-2 1989. pp. 350-353.

[10] Y. Li, B. Vucetic, and Y. Sato, "Optimum Soft-Output Detection for Channels with Intersymbol Interference," *IEEE Transactions on Information Theory*, Vol. 41, No. 3 pp.704-713. May 1995.

[11] J. Hagenauer, E. Offer and L. Papke, "Iterative Decoding of Binary Block and Convolutional Codes," *IEEE Transactions on Information Theory*, Vol. 42, No. 2. pp. 429-445. March 1996.

[12] N. G. Kingsbury and P. J. W. Rayner, "Digital Filtering Using Logarithmic Arithmetic," *Electronics Letters*, Vol. 7, No. 2. pp. 56-58, Jan. 1971.

[13] C. B. Shung, P. H. Siegel, G. Underboeck, H. K. Thapar, "VLSI Architectures for Metric Normalization in the Viterbi Algorithm", *ICC'90*, vol. 4, pp 1723 -8. 1990.

[14] P. Tortelier, D. Duponteil, "Dynamique des métriques dans l'algorithme de Viterbi", *Annales des Télécommunications*, CNET, 45, n° 7-8, pp 377-83, 1990.

[15] P. H. Siegel, C. B. Shung, T. D. Howell, H. K. Thapar, "Exact bounds for Viterbi Detector Path Metric Differences", *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91*. Vol. 2. pp 1093 - 6.

[16] H. Dawid, G. Gehnen, H. Meyr, "Map Channel Decoding: Algorithm and VLSI architecture", *VLSI Signal Processing VI*, 1993., pp. 141-9.

[17] H. Dawid, H. Meyr, "Real-time Algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding", *Personal, Indoor and Mobile Radio Communications, 1995. PIMRC'95. Wireless: Merging onto the Information Superhighway*. Vol. 1, pp. 193 -7.

[18] QUALCOMM INC, "Q-1401 single chip Viterbi decoder", Production information 1987.

[19] E. Boutillon, N. Demassieux, “A generalized precompiling scheme for surviving path memory management in Viterbi decoders”, *ISCAS’93*, New-Orleans, May 1993. vol. 3, pp. 1579-82.

[20] E. Boutillon, “Architecture et implantation VLSI de techniques de modulations codées performantes adaptées au canal de Rayleigh”, Chap. 4, Rapport de thèse n° ENST 95 E 041, déc. 95.