TSG-RAN Working Group 1(Radio) meeting #6
Espoo, Finland 13-16th , July 1999

TSGR1#6(99) 908

Source: **LGIC**

Title: Code Symbol Based Uplink Puncturing Algorithm

---

1. Introduction

The conventional uplink puncturing algorithm was designed on the basis of "satisfying the uniform puncturing of all coded bits on the original sequence before the 1st MIL"[1][2][3]. This concept is appropriate for convolutional code and confirms almost optimal performance in the processing of transport channel multiplexing. But, the conventional puncturing algorithm is not optimal for turbo code and a more efficient design of puncturing algorithm could be possible.

For the downlink, there are 2 puncturing methods competing for turbo code puncturing[4][5][6]. The basic and common considerations of these two algorithms can be summarised as follows.

1) Preventing puncturing of systematic bits
2) Approximately equal puncturing of parity bits of the two encoders
3) Providing uniform puncturing for each parity bit sequence

The above 3 considerations can be simply satisfied for the downlink, since rate matching procedure occurs before the 1st MIL and is operated independent of the 1st MIL. But in the uplink, rate matching procedure occurs after the 1st MIL and must be executed on the '1st MIL interleaved sequence'.

If conventional uplink puncturing algorithm is used for turbo code, there is no guarantee that the above 3 considerations are satisfied, and some performance degradation is inevitable. Therefore, the need for uplink turbo puncturing algorithm arises.

For the uplink, besides the above 3 considerations, there is one more restraint, i.e, every interleaved column sequence must have equivalent number of puncturing. Under these constraints, it seems almost impossible to find an algorithm which satisfies all of the above considerations without modifying the 1st MIL or the output pattern for the turbo encoder. But to our knowledge, it is dangerous to modify the 1st MIL because it will change the overall interleaving gain and for the same reason it is also dangerous to modify the output pattern of turbo encoder severely. That is, there is a possibility of deep fading which impacts mostly on the systematic bits components or parity bits components. This will cause some performance degradation in realistic fading environments.

In this proposal, we propose a new novel uplink puncturing algorithm for turbo code without severely changing the characteristics of 1st MIL which has a good performance gain over the conventional algorithm. This algorithm has a common structure with LGIC's downlink puncturing algorithm[4][5]. Therefore, a unified rate matching scheme can be provided. Even in the case of puncturing for convolutional code, proposed algorithm can be applied without loss of performance.

2. Design of Uplink Puncturing Algorithm for Turbo Code

The design rules imposed on the uplink puncturing algorithm for turbo code are as below.

Code symbol(word) based puncturing process in a view point of the "before the 1st MIL original sequence"
For the uplink algorithm for convolutional code, every code bit has approximately equal importance in the sense of BER performance and uniform puncturing over all "before the 1st MIL original code bit sequence" is appropriate. But for the turbo code, the coded bits of a code symbol have unequal importance depending on whether it is a systematic or parity bits. Therefore the "bit based puncturing" is no more suitable and it is more efficient to use a symbol based approach.

Preventing puncturing of systematic bits
Systematic bits of turbo code are more important than parity bits which means that puncturing of one systematic bit results in more performance degradation than a parity bit.

Equal amount and uniform puncturing of parity bits of two encoders
In order to maximise the BER performance of turbo code, the coding strength of each RSC code must be balanced. Balanced puncturing of parity bits between the two encoders means balanced puncturing of each RSC code.

Equal amount of puncturing for each 1st MIL interleaved column sequence
For the uplink, rate matching algorithm is performed over 1st MIL interleaved sequence and therefore, rate matching must be performed in a way that every column sequence has an equal amount of puncturing. The purpose of uplink rate matching for turbo code is to satisfy the original property of turbo puncturing algorithm in the view point of "before the 1st MIL original code sequence" while applying the equal amount of puncturing over each interleaved sequence.

Providing a unified rate matching algorithm for uplink and downlink
For the simplicity of implementation, it is desirable to use a unified rate matching algorithm for uplink and downlink.

3. The Principle of Proposed Uplink Puncturing Algorithm

Figure 1 shows the problem of applying the conventional uplink rate matching to the turbo code where among the total of 96 code bits, 16 code bits are punctured. In the figure, 'x' means the systematic code bit, 'y' the parity bit from 1st RSC encoder and 'z' from the 2nd RSC encoder. The shaded parts represent the position of puncturing. As you can see, the puncturing of systematic bit is inevitable and it will lead to some performance degradation.

x	y	z	x	y	**z**	x	y
z	x	y	**z**	x	y	z	x
y	**z**	x	y	z	x	y	z
x	y	z	x	y	z	**x**	y
z	x	y	z	**x**	y	z	x
y	z	**x**	y	z	x	y	**z**
x	y	z	x	y	**z**	x	y
z	x	y	**z**	x	y	z	x
y	**z**	x	y	z	x	y	z
x	y	z	x	y	z	**x**	y
z	x	y	z	**x**	y	z	x
y	z	**x**	y	z	x	y	**z**

Figure 1. The problem of applying the conventional uplink rate matching algorithm to turbo code

Figure 2 shows the problem of uplink rate matching algorithm for turbo code in the extreme case of 33.3% puncturing among the total code bits. As you can see in figure 2, each odd column sequence has 8 bit of puncturing, while there is no puncturing on the even number column sequences. This is a violation of the puncturing rule for uplink.

x	**y**	z	x	y	**z**	x	**y**
z	x	y	**z**	x	**y**	z	x
y	**z**	x	**y**	z	x	y	**z**
x	**y**	z	x	y	**z**	x	**y**
z	x	y	**z**	x	**y**	z	x
y	**z**	x	**y**	z	x	y	**z**
x	**y**	z	x	y	**z**	x	**y**
z	x	y	**z**	x	**y**	z	x
y	**z**	x	**y**	z	x	y	**z**
x	**y**	z	x	y	**z**	x	**y**
z	x	y	**z**	x	**y**	z	x
y	**z**	x	**y**	z	x	y	**z**

Figure 2. Establishment of the problem of uplink puncturing for turbo code.

The simple and easy way to avoid hitting the same column is to change the output pattern of the turbo encoder slightly. That is, using a simple switch, the interleaver memory can be written as in the example of figure 3.

x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y

Figure 3. Change of output pattern of turbo encoder for 1st MIL interleaver

This output pattern is obtained by generating the code bit sequence as x, y, z, x, z, y, x, y, z, ……using the simple switch over 2 parity bit y, z. As you can see in the figure, by changing the order of parity bits alternatively, we can avoid the problem of hitting the same column and optimum puncturing pattern for 33.3% can be acquired.

As can be seen in figure 3, by switching the order of output code bit, we can restrict the possible puncturing position to the centre of each code symbol over "before the 1st MIL code symbol sequence". For even-numbered code symbol, we can puncture the parity 'y' and for odd-numbered symbol, the parity 'z'. These are the simple principles of the proposed algorithm.

Using these principles, we can find an algorithm that satisfies all the requirements for turbo code, which are to avoid the puncturing of systematic bits, provide equal amount and uniform puncturing over each parity bit, and provide equal amount of puncturing for each "1st MIL interleaved column sequence".

4. Procedure of Uplink Puncturing Algorithm for Turbo Code

[1st Step]: *Calculation of Shifting Parameter Guaranteeing "the Code-Symbol Based Uniform Puncturing"*

To guarantee the uniform puncturing over 2 parity bit sequences, we can use a similar procedure to the conventional approach for calculating the shifting parameter of each column of 1st MIL interleaver. While the conventional method calculates the shifting parameter based on the code bit space, we modified it to calculate the shifting parameter based on the code symbol space. For these, we divided the code symbols into two groups. One is the group of code symbols for "even numbered code symbol" and the other is the group of code symbols for "odd numbered code symbol". Then to each group, we can apply the similar procedure for calculating the shifting parameter S.

This procedure can be simply described by figure 4. By dividing the over all code symbol space into 2 groups, and assuming that each group of code symbols is written in the interleaver of K/2 columns, we can obtain figure 4. The left figure is for 'even numbered symbol group' while the right is for 'odd-numbered symbol group'. In each column of the two virtual interleaver , there are shaded symbol positions which means "punctured symbol positions". The left figure is for the parity sequences 'y' and the right figure is for the parity sequences 'z'. The bold number of the 1st row for each virtual interleaver represents the actual column number of original interleaver. That is, the 1st column of the left interleaver represents the 1st column of the original MIL interleaver and the 2nd column represents the 7th column of the original MIL interleaver, and vice versa. The mapping of virtual interleaver column index to the original interleaver column index can be simply described by the next equation

$$Q(k) \quad = \quad (6k+1) \quad \mathrm{mod} \quad K \;:\; k = 0,1,2,\cdots\frac{K}{2}-1 \quad \text{(mapping rule for even numbered code symbol)}$$

$Q(k) \quad = \quad (6k+4) \quad \mathrm{mod} \quad K \ : \ k = 0,1,2,\cdots \frac{K}{2} - 1$ (mapping rule for odd numbered code symbol)

1	7	5	3
0	2	4	6
8	10	12	14
16	18	20	22
24	26	28	30
32	34	36	38
40	42	44	46
48	50	52	54
56	58	60	62
64	66	68	70
72	74	76	78
80	82	84	86
88	90	92	94

4	2	0	6
1	3	5	7
9	11	13	15
17	19	21	23
25	27	29	31
33	35	37	39
41	43	45	47
49	51	53	55
57	59	61	63
65	67	69	71
73	75	77	79
81	83	85	87
89	91	93	95

Figure 4. calculation of shifting parameter for each column via virtual interleaver.

Then, the remaining problem is explicit. We can find the shifting parameter *S* for each column of the virtual interleaver to guarantee the uniformity in the original 1st MIL interleaver. The procedure can be described as follows.

Start of Procedure A

If among the total number of N_c code bits, P bits are to be punctured. Then next parameter can be defined.

$N = \left\lfloor \frac{N_c}{3} \right\rfloor$: $\lfloor x \rfloor$ represents the largest integer which does not exceed the number *x*

$N_i = N - P$

$q = \left\lfloor \frac{N}{|N_i - N|} \right\rfloor$: q represents the average puncturing distance over code symbol space.

From the value of q, we can find the shifting parameter *S* guaranteeing the overall uniformity over "before the 1st interleaved code sequence" as follows.

```
if($q \leq 2$){
        for($k$=0; $k$<2/$K$; $k$++) {
                if(($k$%2)==0){
                        $S[R[(6k+1) \bmod K]] = 0$;
                        $S[R[(6k+4) \bmod K]] = 0$;
                }
                else{
                        $S[R[(6k+1) \bmod K]] = 1$;
                        $S[R[(6k+4) \bmod K]] = 1$;
                }
         }
}
else{
        if(($q$%2)=1)
```

```
            $q' = q - \frac{G.C.D(q, K/2)}{K/2}$;  to avoid hitting the same column
    else    $q' = q$
    for($i$=0; $i < \frac{K}{2}$; $i$++) {
            $k = \lceil i * q' \rceil \; \% \; \frac{K}{2}$;
            $S[R[(6k+1) \bmod K]] = \lceil i * q' \rceil \text{ div } \frac{K}{2}$;
            $S[R[(6k+4) \bmod K]] = \lceil i * q' \rceil \text{ div } \frac{K}{2}$
     }
}
```

<u>End of Procedure A</u>

In the above procedure, R [] means the mapping pattern of the 1st MIL interleaver.

<u>[2nd Step] *: Applying the puncturing algorithm using the shift parameter from procedure A*</u>

The 2nd step of the proposed uplink puncturing algorithm is to perform the rate matching using the shift parameter of each column calculated in the 'procedure A'. In the process of puncturing using the proposed algorithm, each column has a unique puncturing position among the 3 code bits. For example, every 3rd bit of 3 coded bits may be punctured for 0th column and every 1st bit of 3 coded bits for 1st column. This position can be described by simple equation.

```
    for($k$=0; $k<K$; $k$++) {
            pos = ($K$ ( $R$ [$k$]-1 ) + 11) % 3;
            puncturing 3$m$ – pos where $m$ = 1, 2, …, $\left\lfloor \frac{N_c}{3} \right\rfloor$
    }
```

Then, proposed uplink puncturing algorithm for turbo code can be described as follows.

<u>Start of Puncturing Procedure</u>

```
if($K$=1)  *LGIC Puncturing algorithm for downlink*
if($K \geq 2$) {
    $S_0 = \{d_{N_1}, d_{N_2}, \cdots d_{N_c}\}$ : set of $N_C$ data bits for each column
    $N_i$ : *symbol number after puncturing*
    $N = \lfloor N_c / 3 \rfloor$
    *k : column index  k=0,1,2,3,….,K-1 (K : Column number of 1st MIL)*
    *if puncturing is to be performed*
            $y = N - N_i$
```

```
        $e = (2*S(k) * y + N) \bmod 2N$        – initial error $e_{offset}$
        if($e$=0)  $e = 2N$
        $m = 1$                                -- index for current symbol
        do while $m <= N$
            $e = e - 2 * y$                        -- update error
            if $e <= 0$ then                   -- check if symbol number m should be punctured
                puncture bit $3m - ((K(R[k]-1)+11) \bmod 3)$ from set $S_0$
                $e = e + 2*N$                      -- update error
            end if
            $m = m + 1$                        --  index for next symbol
        end do
    end if
```

<u>End of Puncturing Procedure</u>

Figure 5 is an example of punctured position when the algorithm described above is applied.

x	y	z	x	z	y	x	**y**
z	x	**z**	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	**y**	z	x	**z**	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	**y**	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	**y**	z	x	**z**	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	**y**
z	x	**z**	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	**y**	z	x	**z**	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	**y**	z	x
z	y	x	y	z	x	z	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	**y**	z	x	**z**	y
x	y	z	x	z	y	x	y
z	x	z	y	x	y	z	x
z	y	x	y	z	x	z	y

Figure 5. example of the proposed turbo puncturing algorithm

As you can see in figure 5, all of the requirements are satisfied when the proposed algorithm is applied.

5. Applying the Proposed Algorithm to Convolution Code

In our investigation of puncturing algorithm for downlink and uplink, we have found out that there exist some patterns of puncturing which ensure better performance not only for the case of turbo code but also for the case of convolutional code. These patterns can not be obtained through the conventional ‘bit based approach’. But in the case of ‘code symbol based approach’, there are possibilities of finding certain patterns for puncturing.

For example, in the 1/3 convolutional code, we can puncture only the ‘output 0’ which is generated from polynomial ‘753’ for each code symbol, or we can puncture the ‘output1’ and ‘output 2’ alternatively.

Through the extensive computer simulation for downlink, we have found out that the above 2 puncturing patterns show similar or in some cases, better performance compared with the conventional algorithm when applied to the convolutional code.

These two puncturing patterns can be obtained also in the uplink, without changing the puncturing algorithm itself.

That is, if we want to puncture only the ‘output 0’, we can generate the code bit sequence in the form of y, x, z, y, x, z … . There is no need for switching in this case. Then, we can calculate the shifting parameter using the similar algorithm of procedure A. But for this case, there is no need to divide the code symbol space into 2 groups. The procedure can be modified as follows.

Start of Procedure B : Calculation of Shifting Parameter for Convolutional Code

The definition of all the parameters is the same as that of procedure A

From the value of q, we can find the shifting parameter S guaranteeing the overall uniformity over “before the 1st interleaved code sequence” as follows.

```
if($q \le 2$){
    for($k$=0; $k$<$K$; $k$++) {
        if(($k$%2)=0)        $S[R[(3k+1) \bmod K]] = 0$;
        else                 $S[R[(3k+1) \bmod K]] = 1$;
    }
}
else{
    if(($q$%2)=1)
        $q' = q - \frac{G.C.D(q,K)}{K}$ ;  to avoid hitting the same column
    else    $q' = q$
    for($i$=0; $i$< $K$ ; $i$++) {
        $k = \lceil i*q' \rceil \% K$ ;
        $S[R[(3k+1) \bmod K]] = \lceil i*q' \rceil \text{ div } K$
    }
}
```

End of Procedure B : Calculation of Shifting Parameter for Convolutional Code

Then, using the shifting parameter S from procedure B and puncturing procedure which is common for turbo code and convolution code, uplink puncturing for convolutional code can be performed.

If the alternative puncturing of ‘output 1’ and ‘output 2’ among the code symbols is preferred, we can generate the convolutional code sequence in the same way as in the case of turbo code sequence. Then, using procedure A, shifting parameter of each column can be obtained and the proposed uplink puncturing algorithm can be used without changing the algorithm at all.

The performance comparisons between the conventional and proposed approach will be described in section 8.

6. Universal Rate Matching Algorithm for Uplink

In this section, we propose an universal rate matching algorithm for uplink. As has been stated, code symbol based uplink rate matching algorithm can be applied to the convolutional code without loss of performance and in some cases can provide better performance compared with the conventional algorithm. The basic routine for puncturing can be applied to the convolutional code without changing the algorithm itself. We only need to change the shifting parameter for each column and then apply the identical puncturing procedure in the case of convolutional code. The universal puncturing algorithm for uplink can be described as follows.

```
if(column number for 1st MIL K=1)
    downlink LGIC puncturing algorithm

else {
    if(convolutional code)
        calculate shifting parameter S using procedure A or procedure B
    else if(turbo code)
        calculate shifting parameter S using procedure A


    Perform the puncturing process using the common puncturing algorithm for uplink
}
```

7. Unification of Puncturing Algorithm for Uplink and Downlink

The puncturing procedure described in section 4 is very similar to the procedure for downlink. The only difference is the position of punctured bits in one symbol, therefore uplink and downlink puncturing algorithm can be implemented by one common hardware, with some parameter changes**.** Figure 6 shows the unified rate matching block.

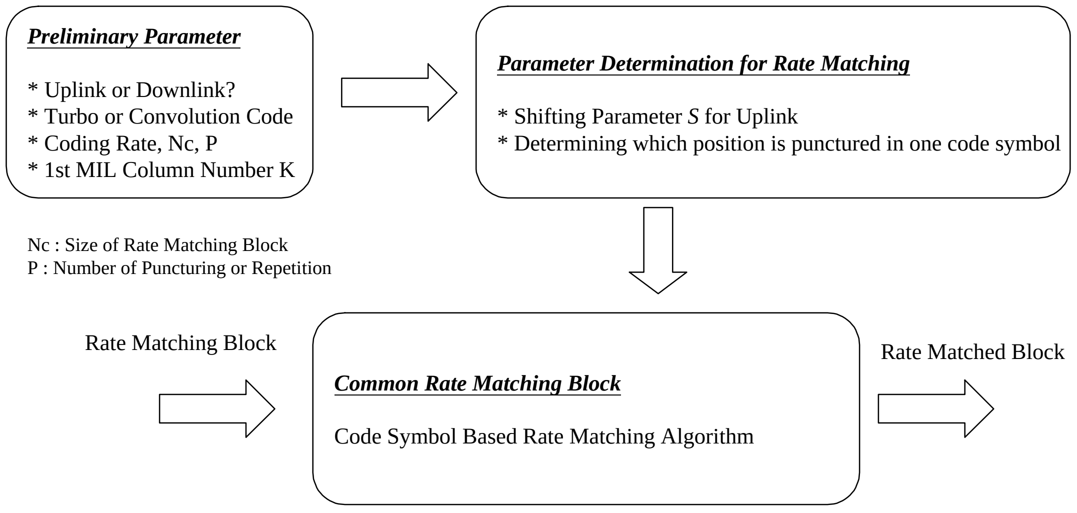
Preliminary Parameter
* Uplink or Downlink?
* Turbo or Convolution Code
* Coding Rate, Nc, P
* 1st MIL Column Number K
Parameter Determination for Rate Matching
* Shifting Parameter S for Uplink
* Determining which position is punctured in one code symbol
Nc : Size of Rate Matching Block
P : Number of Puncturing or Repetition
Rate Matching Block
Common Rate Matching Block
Code Symbol Based Rate Matching Algorithm
Rate Matched Block

Figure 6. unified rate matching block for uplink and downlink.

As you can see in figure 6, the basic rate matching block is common to the all cases. In other words, the basic rate matching block can be applied without change whether convolutional code or turbo code is used or whether it is for downlink or uplink. All you need to do is to determine the parameters for rate matching block from the preliminary input parameters, which can be an indicator for uplink or downlink, an indicator for turbo or convolutional code, etc. In figure 6, the 'parameter determination block' sets up all the parameters necessary in the rate matching block. For example, the parameters such as the shifting parameter *S* for uplink, the puncturing position in one code symbol, etc are determined in this block. Using these parameters, rate matching procedure starts.

8. Simulation Results

Simulation conditions for turbo code are as below.

- Interleaver depth is 964 and 5116 for turbo code where internal interleaver is PIL.
- Constraint length of the constituent code is 4.
- Conventional termination method is applied.
- A full MAP with floating point implementation is used for the decoding of the constituent encoders
- Iteration number is 4.
- The number of column for 1st MIL is 8
- At a BER of 10^{-5}, at least 100 frame errors have to be counted
- Simulations are carried out in an AWGN channel
- Conventional puncturing is performed for comparison purpose
- Puncturing number : 576 bit for 964 interleaver depth and 3072 bit for 5116 interleaver depth.

Figure 7 shows the performance of BER and FER in the case of puncturing for turbo code when the internal interleaver depth is 964 and 576 bit puncturing is applied. As you can see in the figure, performance gain of 0.08~0.1dB can be achieved by using the proposed algorithm.

Figure 8 shows the performance of BER and FER in the case of puncturing for turbo code when the internal interleaver depth is 5116 and 3072 bit puncturing is applied. As can be seen in the figure, performance gain of more than 0.1dB can be achieved via proposed algorithm.

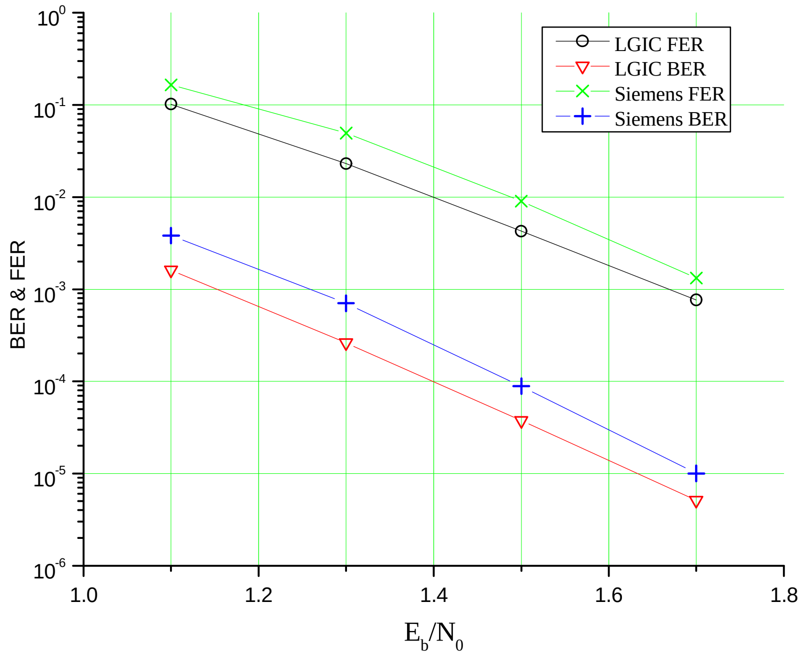

Figure 7. puncturing for turbo code (interleaver depth : 964, 576 bit puncturing)

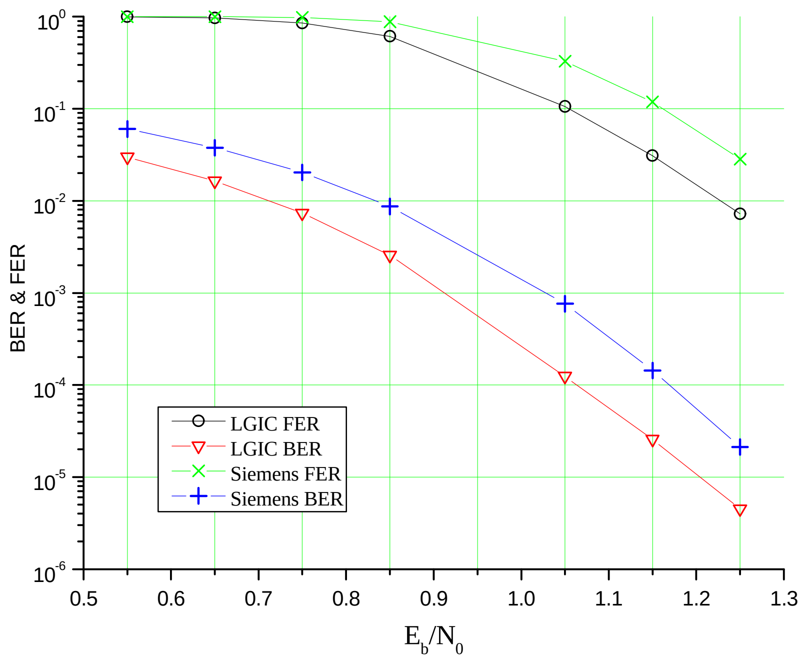

Figure 8. puncturing for turbo code (interleaver depth : 5116, 3072 bit puncturing)

Simulation conditions for convolutional code are as below.

- Frame length is 160 and constraint length of the convolutional code is 9.
- 8 tail bits are used for termination.
- Decoding algorithm : full path memory Viterbi algorithm with the tail information.
- The number of column for 1st MIL is 8
- At a BER of 10^{-5} , at least 100 frame errors have to be counted
- Simulations are carried out in an AWGN channel
- Puncturing number : 56 bits and 96 bits.
- LGIC 1 : puncturing only the ‘output 0’ bit .
- LGIC 2: puncturing alternatively the ‘output 1’ and the ‘output 2’.

Figure 9 shows the performance of BER and FER in the case of puncturing for convolutional code when the frame length is 160 and 56 bit puncturing is applied. As you can see in the figure, there is no difference in performance in the range of BER $>10^{-4}$. But in the range of BER $<10^{-4}$, the proposed code symbol based approach provides a little better performance.

Figure 10 shows the performance of BER and FER when the frame length is 160 and 96 bit puncturing is applied. As you can see in the figure, there is no difference in performance in the BER and FER.

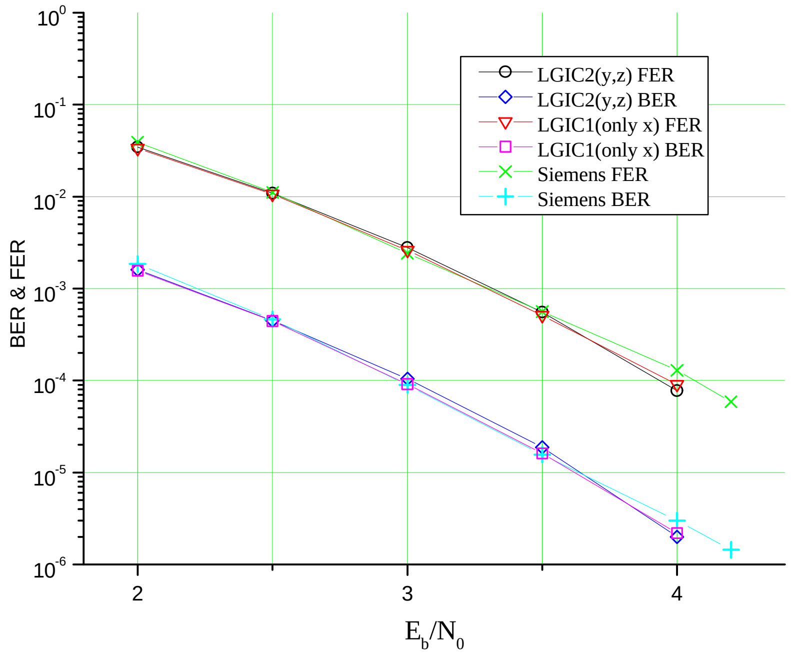

Figure 9. puncturing for convolutional code (frame length : 160, 56 bit puncturing)

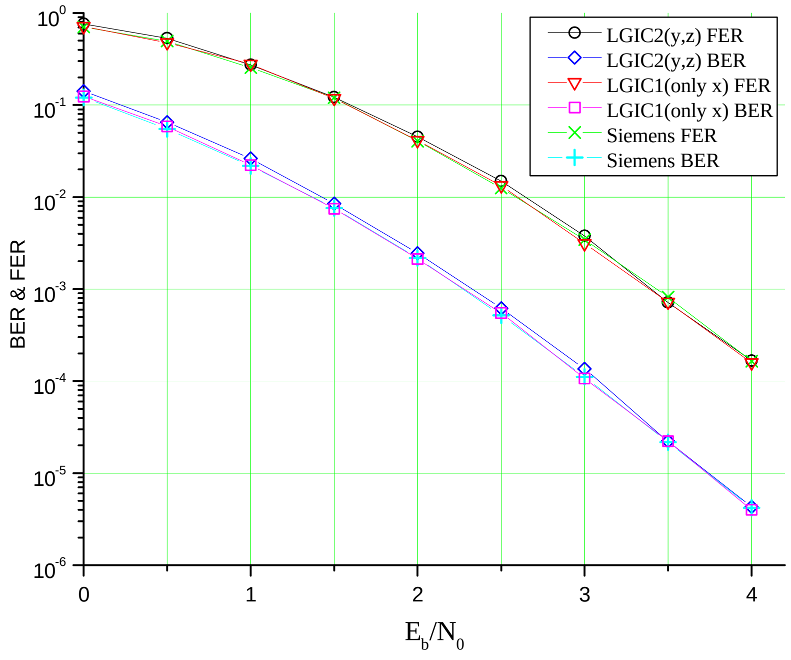

Figure 10. puncturing for convolutional code (frame length : 160, 96 bit puncturing)

9. Conclusion

To enhance the performance of the uplink puncturing algorithm for turbo code, a novel uplink puncturing algorithm is proposed which satisfies all of the requirements for the turbo code puncturing. The proposed puncturing algorithm is simple and has superior performance over the conventional algorithm in the case of turbo code. And even in the case of uplink puncturing for convolutional code, the proposed puncturing scheme can be applied without the loss of performance and, in some case, even provides better performance compared with the conventional algorithm.

Because the proposed ‘code symbol based’ rate matching algorithm for uplink can be used for downlink in a similar manner through some parameter changes, a unified rate matching algorithm for uplink and downlink can be provided.

In conclusion, it is believed that the proposed method is a strong candidate for universal puncturing algorithm.

10. Reference

[1] 3GPP TSG RAN WG1 Multiplexing and Channel Coding(FDD) TS 25.212 V1.1.0 (1996. 06)

[2] 3GPP TSG RAN WG1 R1-99203 Optimised Rate Matching After Interleaving, Siemens.

[3] 3GPP TSG RAN WG1 R1-99703 Text Proposal for Optimised Puncturing, Siemens.

[4] 3GPP TSG RAN WG1 R1-99338 Puncturing Algorithm for Turbo Code, LGIC.

[5] 3GPP TSG RAN WG1 R1-99654 Comparison of Downlink Puncturing Algorithms, LGIC.

[6] 3GPP TSG RAN WG1 R1-99388 Optimised Puncturing Scheme for Turbo Coding, Fujitsu.

Contact info : mailto: youngwooy@lgic.co.kr