A Fast ACSU Architecture for Viterbi Decoder Using *T*-Algorithm

Jinjin He, Huaping Liu, *Senior Member, IEEE*, and Zhongfeng Wang*, *Senior Member, IEEE*
School of EECS, Oregon State University, Corvallis, Oregon, 97330, email:{hejin, hliu} eecs.orst.edu.
*Broadcom Corporation, Irvine, CA, 92618 USA, email: zfwang@broadcom.com.

***Abstract*— Modern digital communication systems usually employ convolutional codes with large constraint length for good decoding performance, which leads to large complexity and power consumption in Viterbi decoders. It is essential to use *T*-algorithm in Viterbi decoders to prune significant portions of the trellis states to dramatically reduce power consumption. However, the operation of searching for the best path metrics in the add-compare-select loop in *T*-algorithm significantly limits the clock speed. In this paper, we propose an efficient architecture based on pre-computation for Viterbi decoders incorporating *T*-algorithm. Through optimization at both algorithm level and architecture level, the new architecture greatly shortens the long critical path introduced by the conventional *T*-algorithm. The design example provided in this work demonstrates more than twice improvement in clock speed with negligible computation overhead while maintaining decoding performance.**

I. INTRODUCTION

Convolutional coding is one of the widely used techniques for correcting errors in digital communication systems. The error-correcting capability of a convolutional code is related to the width of correlated bits, which is referred to as constraint length k. Convolutional codes with a larger k could in general provide better performance. For decoding of convolutional codes, maximum likelihood (ML) algorithms are typically employed, among which the Viterbi algorithm (VA) [1] achieves ML performance with low computation complexity.

A typical functional diagram of the corresponding Viterbi decoder (VD) is shown in Fig. 1. First, branch metrics (BMs) are calculated from the received symbols. Then, BMs are fed into the add-compare-select unit (ACSU) that recursively computes the path metrics (PMs) and outputs decision bits for each possible state transition. After that, the decision bits are stored in and retrieved from the survivor-path memory unit (SMU) in order to decode the source bits along the final survivor path. The PMs of the current iteration are stored in the PM unit (PMU) and read out for use in the next iteration.

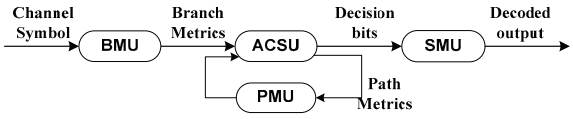

Figure 1. Functional diagram of a Viterbi Decoder.

For VD implementation, research efforts have been focused on the ACSU and SMU. ACSU implementation is critical because the feedback loop makes it the bottleneck for high speed applications. Furthermore, as the constraint length K increases, the computation complexity and power consumption increase exponentially. To deal with these drawbacks, it is a common practice to use *T*-algorithm [2], [3] in VD for power saving purpose. In the *T*-algorithm, a threshold T is set and the difference between each PM and the optimal one is calculated. The algorithm compares the differences with T; only the states with a difference less than T survive and are used for the calculation in the next cycle. Since the process involves the searching of the optimal PM in the ACS loop, clock speed of the entire design will decrease.

To achieve high speed, it is possible to implement a $2^{(k-1)}$ inputs comparator with fully parallel architecture. However, it will cause significant hardware overhead, which conflicts with the design goal of less computation and low power consumption. Several works [4]-[6] have proposed new schemes for high speed *T*-algorithm implementation. All these schemes use an estimated (or approximated) optimal PM derived from the optimal BM instead of finding the accurate value in each cycle. Also, in these methods, compensation schemes are introduced to ensure that the estimated value is not drifting too far away from the accurate one.

These methods combined with compensation algorithms have shown good results on low-rate ($1/R$, R =2,3,4…) codes in [4]-[6]. However, we have observed from our simulation results that, as the code rate increases, the bit-error-rate (BER) performance degrades. This is mainly caused by the drifting error of the estimated optimal PM value, which increases as the code rate increases. When these schemes are applied on high-rate ((R -1)/ R, $R > 2$)) codes, serious problems (losing the entire survivor path) that significantly affect the BER performance arise. Moreover, these methods usually require extra parameters for the compensation algorithms, which can only be obtained from simulations and cannot be adjusted based on real-time channel information, making it less attractive in practical implementation.

In this paper, we propose a new architecture for the Viterbi decoder using *T*-algorithm. Unlike existing works such as [4]-[6], an accurate optimal PM is guaranteed to be found at each cycle and no extra parameters are needed. Since the optimal PM is accurate in the proposed architecture, the new architecture keeps the same BER performance as the conventional *T*-algorithm, and is well suited for high-rate codes. Meanwhile, pre-computation combined with pipelining greatly shortens the long critical path. It will be shown in Section III that the critical path of the new scheme could reach the theoretical iteration bound.

The remainder of the paper is organized as follows. Section II presents the general idea of the proposed pre-computation scheme. The example of a high-rate code and the details of the design are discussed in Section III. Section IV presents the simulation and synthesis results, followed by conclusion in Section V.

II. THE PRE-COMPUTATION ALGORITH

The basic idea of pre-computation is as follows. Consider a VD for a convolutional code with the constraint length of k, where each state receives p candidate paths. If the branch metrics are calculated based on the Euclidean distance, the optimal PM becomes the minimum value of all the PMs.

$$
\begin{aligned}
PM_{opt}(n) &= min\{\, PM_0(n), PM_1(n), \ldots\ldots, PM_{2^k-1}(n) \,\} \\
&= min\{ \\
&\qquad min\,[PM_{0,0}(n\text{-}1) + BM_{0,0}(n), \\
&\qquad\quad PM_{0,1}(n\text{-}1) + BM_{0,1}(n), \ldots\ldots, \\
&\qquad\quad PM_{0,p}(n\text{-}1) + BM_{0,p}(n)], \\
&\qquad min\,[PM_{1,0}(n\text{-}1) + BM_{1,0}(n), \\
&\qquad\quad PM_{1,1}(n\text{-}1) + BM_{1,1}(n), \ldots\ldots, \\
&\qquad\quad PM_{1,p}(n\text{-}1) + BM_{1,p}(n)], \\
&\qquad \ldots\ldots\ldots\ldots\ldots\ldots, \\
&\qquad min\,[PM_{2^k-1,0}(n\text{-}1) + BM_{2^k-1,0}(n), \\
&\qquad\quad PM_{2^k-1,1}(n\text{-}1) + BM_{2^k-1,1}(n), \ldots\ldots, \\
&\qquad\quad PM_{2^k-11,p}(n\text{-}1) + BM_{2^k-1,p}(n)] \\
&\quad \} \\
&= min\{\, PM_{0,0}(n\text{-}1) + BM_{0,0}(n), \\
&\qquad PM_{0,1}(n\text{-}1) + BM_{0,1}(n), \ldots\ldots, \\
&\qquad PM_{0,p}(n\text{-}1) + BM_{0,p}(n), \\
&\qquad PM_{1,0}(n\text{-}1) + BM_{1,0}(n), \\
&\qquad PM_{1,1}(n\text{-}1) + BM_{1,1}(n), \ldots\ldots, \\
&\qquad PM_{1,p}(n\text{-}1) + BM_{1,p}(n), \\
&\qquad \ldots\ldots\ldots\ldots\ldots\ldots, \\
&\qquad PM_{2^k-1,0}(n\text{-}1) + BM_{2^k-1,0}(n), \\
&\qquad PM_{2^k-1,1}(n\text{-}1) + BM_{2^k-1,1}(n), \ldots\ldots, \\
&\qquad PM_{2^k-11,p}(n\text{-}1) + BM_{2^k-1,p}(n)\,\}. \qquad (1)
\end{aligned}
$$

The trellis butterflies for a VD usually have a symmetric structure. In other words, the states can be grouped into m clusters, where all the clusters have the same number of states and all the states in the same cluster will be extended by the same BMs. Thus, Eq. (1) can be re-written as

$$
\begin{aligned}
&PM_{opt}(n) = min\{ \\
&min\ (\text{PMs } (n\text{-}1) \text{ in cluster } 1) + min(\text{BMs}(n) \text{ for cluster } 1), \\
&min\ (\text{PMs } (n\text{-}1) \text{ in cluster } 2) + min(\text{BMs}(n) \text{ for cluster } 1), \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots, \\
&min\ (\text{PMs } (n\text{-}1) \text{ in cluster } m) + min(\text{BMs}(n) \text{ for cluster } m) \\
&\}. \qquad (2)
\end{aligned}
$$

The min(BMs) for each cluster can be easily obtained from the BMU, and the min(PMs) at time n-1 in each cluster can be pre-calculated at the same time when the ACUS is updating the new PMs for time n. Therefore, a look-ahead architecture is formed here to calculate the accurate optimal PM at time n. Theoretically, when we continuously decompose PMs(n-1), PMs(n-2),……, the pre-computation scheme can be extended to q steps, where q is any positive integer that is less than n. Hence, $PM_{opt}(n)$ can be calculated directly from PMs(n-q) in q cycles.

For low-rate convolutional codes, pre-computation is usually inefficient because the number of states in the VD is much greater than that of BMs. In this case, at least 4 steps of pre-computation are needed to maintain an acceptable clock speed, which will cause large amount of hardware and computation overhead. However, for high-rate codes, the number of BMs is also large and each state receives more than 2 candidate paths. In this case, one to two steps of pre-computation are enough since regular update of new PMs also takes long time. In the next section, we show an example of a rate-¾ code that employs 1 or 2 steps of pre-computation. The clock speed can approach the fastest value achievable in theory. In addition, the BER performance of the proposed scheme is the same as that of the conventional T-algorithm.

III. THE PRE-COMPUTATION ARCHITECTURES

High-rate convolutional codes are commonly employed for spectrum-constraint applications. For example, the rate-¾ code shown in Fig. 2 is used in a 4-dimensional trellis coded modulation (4-D TCM) for Space Data Systems [7].

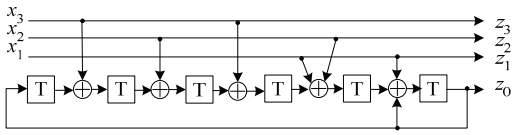

Figure 2. Rate ¾ convolutional encoder.

Although there are only 64 states in the corresponding VD for the code shown in Fig. 2, the number of BMs is 16 and each state receives 8 candidate paths. The computation complexity of this VD is equivalent to that of a rate-½ code with 256 states. Therefore, applying T-algorithm can effectively reduce the overall computation complexity. The BER performance and computational complexity of the VD employing T-algorithm with different threshold are shown in Fig. 3 and Fig. 4, respectively. The simulation is set for a decoder dealing with rate 11/12 4-D TCM signals [7] in an additive white Gaussian noise (AWGN) channel. It is shown in Fig. 3 that the threshold "Tpm" can be reduced to 0.3 with less than 0.1 dB performance loss compared with the ideal Viterbi algorithm.

In the above discussion, computational complexity refers to the number of average candidate paths generated by ACSU each cycle. For a regular VD, there are 512 (i.e., 64×8) paths. It can be observed from Fig. 4 that, as we lower down threshold "Tpm", the number of average enabled states as well as the enabled additions is dropping down. At the high SNR region (SNR ≥ 12 dB), the number of enabled additions can be reduced to 1/10 of that for a regular VD (when Tpm = 0.3), which indicates a dramatic reduction of the power consumption. Although the implementation of T-algorithm itself will introduce extra operations, compared with the saved computation, it is really a small portion.

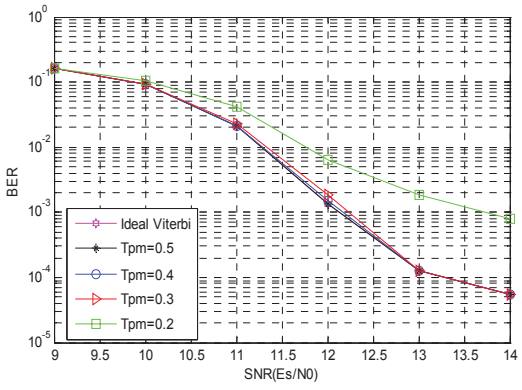

Figure 3. BER performance of T-algorithm.

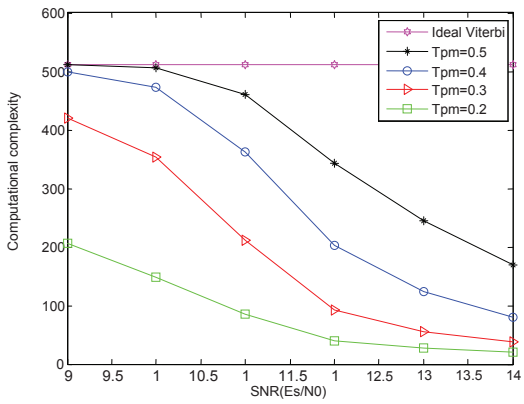

Figure 4. Computational complexity of *T*-algorithm.

A crucial issue with implementing *T*-algorithm is how to quickly find out the optimal PM (the minimum value).

The shortest critical path we can achieve is from the regular ACSU without *T*-algorithm. That's the amount of time each state needs to update its state value, as shown below.

$$T_{full\_trellis} = T_{adder} + T_{8-input\_comparator}.$$

A fully parallel 8-input comparator needs 28 adders and a large look-up table. To achieve a balanced trade-off between hardware area and clock speed, all comparators are designed to process at most 4 data in parallel. Comparators dealing with more inputs are built up with 2-input or 4-input comparators. The critical path now becomes

$$T_{full\_trellis} = T_{adder} + T_{4-in\_comp} + T_{2-in\_comp}. \quad (3)$$

Next, let us consider the VD with *T*-algorithm. The general functional diagram is shown in Fig. 5, where *T*-algorithm is implemented in the "PM purge algorithm" unit (PPAU).

In a VD with conventional *T*-algorithm implementation, the optimal PM is calculated from the 64 newly updated PMs. To find the minimum value of the 64 PMs, we use an architecture consisting of 3-stage 4-input comparators as shown in Fig. 6.

The critical path for the conventional *T*-algorithm implementation is computed by Eq. (4).

$$\begin{aligned} T_{conv\_T-alg} &= T_{adder} + T_{4-in\_comp} + T_{2-in\_comp} \\ &\quad + 3T_{4-in\_comp} + T_{adder} + T_{2-in\_comp} \\ &= 2T_{adder} + 4T_{4-in\_comp} + 2T_{2-in\_comp} \end{aligned} \quad (4)$$

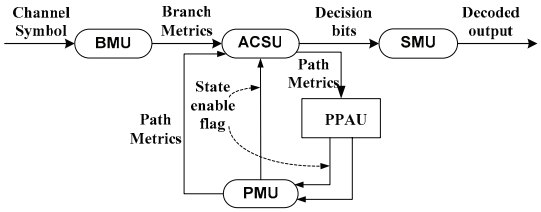

Figure 5. Functional block of a Viterbi decoder with *T*-algorithm.

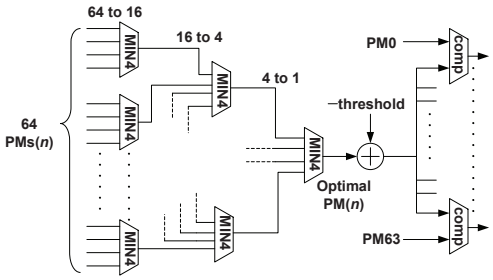

Figure 6. The implementation of PPAU for conventional *T*-algorithm.

The scheme proposed in [6] is called the SPEC-*T* algorithm, where it is assumed the value of (PM_{opt} + threshold) can be obtained during the period when ACSU updates new PMs. The only extra calculation for SPEC-*T* algorithm is the comparison between the PM_{opt} - threshold and all the PMs. Therefore, the critical path for the SPEC-*T* scheme is greatly shortened as shown in Eq. (5).

$$\begin{aligned} T_{SPEC-T\_alg} &= T_{adder} + T_{4-in\_comp} + T_{2-in\_comp} + T_{2-in\_comp} \\ &= T_{adder} + T_{4-in\_comp} + 2T_{2-in\_comp}. \end{aligned} \quad (5)$$

T_{SPEC-T_alg} is also the iteration bound we can get for VD when *T*-algorithm is employed. The functional block of SPEC-*T* algorithm is slightly different from the one shown in Fig.3, where the minimum BM is sent to the PPAU from the BMU. Since the estimated optimal PM is calculated each cycle, an accurate optimal PM is also needed every 6 to 7 cycles to compensate for the estimated one. For example, at time slot *n*, the decoder memorizes $PM_{opt\_esti}(n)$ and PMs(*n*). After 7 cycles, $PM_{opt\_accu}(n)$- $PM_{opt\_esti}(n)$ is added to $PM_{opt\_esti}(n+7)$. The problem with this compensation scheme is that the error between PM_{opt_esti} and PM_{opt_accu} accumulates over at least 7-cycles due to the inherent delay of the scheme itself.

A. One-step pre-computation

For the convenience of discussion, we define the left-most register in Fig. 2 as the most-significant-bit (MSB) and the right-most register as the least-significant-bit (LSB). The 64 states and PMs are labeled from 0 to 63.

A careful study reveals that the 64 states can be partitioned in two groups: odd-numbered PMs (when the LSB is '1') and even-numbered PMs (when the LSB is '0'). The odd PMs are all extended by odd BMs (when Z_0 is '1') and the even PMs are all extended by even BMs (when Z_0 is '0'). The minimum PM becomes:

$PM_{opt}(n)$= *min*{
min (even PMs (*n*-1)) + *min*(even BMs(*n*)),
min (odd PMs (*n*-1)) + *min*(odd BMs(*n*))
}.

The functional diagram of the 1-step pre-computation scheme is shown in Fig. 7. Notice that, in Fig. 5, the PPAU have to wait for the new PMs from the ACSU to calculate the optimal PM, while in Fig. 7 the optimal PM is calculated directly from PMs in the previous cycles at the same time when the ACSU is calculating the new PMs. The details of the PPAU are shown in Fig. 8.

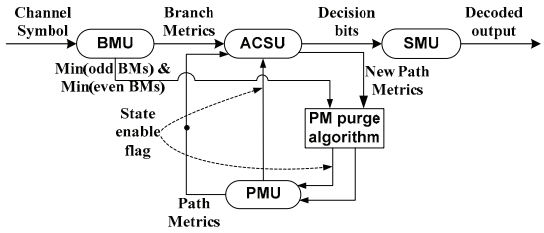

Figure 7. Functional block of a Viterbi decoder with 1-step pre-computation *T*-algorithm.

The critical path of the 1-step pre-computation scheme is

$$T_{1-step\_pre\_T} = 2T_{adder} + 2T_{4-in\_comp} + 3T_{2-in\_comp}, \quad (6)$$

which is much shorter than that in Eq. (4). Note that compared with Fig. 6, the hardware overhead of the 1-step pre-computation scheme is about 4 adders, which is negligible. Compared with the SEPC-*T* algorithm, however, the critical path of the 1-sept pre-computation scheme is still long. In order to further shorten the critical path, we explore the 2-step pre-computation design next.

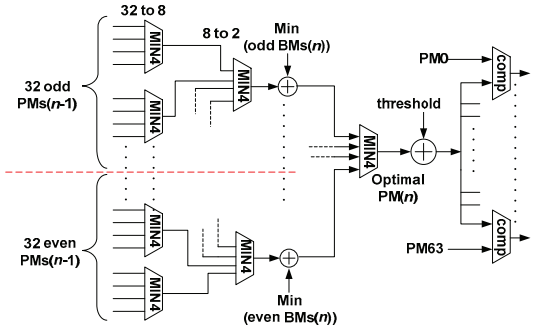

Figure 8. Implementation of 1-step pre-computation *T*-algorithm.

B. Two-step pre-computation

We again need to analyze the trellis transition of the original code. In the 1-step pre-computation architecture, we have pointed out that for the particular code shown in Fig. 2, odd-numbered states are extended by odd BMs, while even-numbered states are extended by even BMs. Furthermore, the even states all extend to states with higher indices (the MSB in Fig. 2 is '1') in the trellis transition, while the odd states extend to states with lower indices (the MSB is '0' in Fig. 2). This information allows us to obtain the 2-step pre-computation data path. This process is straightforward, although the mathematical details are tedious. For clarity, we only provide the main conclusion here.

The states are further grouped into 4 clusters as described by Eq. (7). The BMs are categorized in the same way and are described by Eq. (8).

$$\begin{aligned} \text{cluster0} &= \{\text{PM}_m \mid 0 \le m \le 63, m \bmod 4 = 0\} \\ \text{cluster1} &= \{\text{PM}_m \mid 0 \le m \le 63, m \bmod 4 = 2\} \\ \text{cluster2} &= \{\text{PM}_m \mid 0 \le m \le 63, m \bmod 4 = 1\} \\ \text{cluster3} &= \{\text{PM}_m \mid 0 \le m \le 63, m \bmod 4 = 3\} \end{aligned} \quad (7)$$

$$\begin{aligned} \text{BMG0} &= \{\text{BM}_m \mid 0 \le m \le 15, m \bmod 4 = 0\} \\ \text{BMG1} &= \{\text{BM}_m \mid 0 \le m \le 15, m \bmod 4 = 2\} \\ \text{BMG2} &= \{\text{BM}_m \mid 0 \le m \le 15, m \bmod 4 = 1\} \\ \text{BMG3} &= \{\text{BM}_m \mid 0 \le m \le 15, m \bmod 4 = 3\} \end{aligned} \quad (8)$$

The optimal PM at time n is calculated as

$$\begin{aligned} \text{PM}_{opt}(n) = min\ [& \\ & min\ \{min\ (\text{cluster0}\ (n\text{-}2)) + min\ (\text{BMG0}(n\text{-}1)), \\ & \quad min\ (\text{cluster1}\ (n\text{-}2)) + min\ (\text{BMG1}(n\text{-}1)), \\ & \quad min\ (\text{cluster2}\ (n\text{-}2)) + min\ (\text{BMG3}(n\text{-}1)), \\ & \quad min\ (\text{cluster3}\ (n\text{-}2)) + min\ (\text{BMG2}(n\text{-}1)) \\ & \quad \} + min\ (\text{even BMs}(n)), \\ & min\ \{min\ (\text{cluster0}\ (n\text{-}2)) + min\ (\text{BMG1}(n\text{-}1)), \\ & \quad min\ (\text{cluster1}\ (n\text{-}2)) + min\ (\text{BMG0}(n\text{-}1)), \\ & \quad min\ (\text{cluster2}\ (n\text{-}2)) + min\ (\text{BMG2}(n\text{-}1)), \\ & \quad min\ (\text{cluster3}\ (n\text{-}2)) + min\ (\text{BMG3}(n\text{-}1)) \\ & \quad \} + min\ (\text{odd BMs}(n)) \\ & \quad]. \end{aligned} \quad (9)$$

An intuitive illustration of the process is shown in Fig. 9. The "MIN 16" unit for finding the minimum value in each cluster is constructed with 2 stages of 4-input comparators.

Calculating $\text{PM}_{opt}(n)$ from $\text{PM}(n\text{-}2)$ means that the calculation can be completed within 2 cycles . Thus, the process is pipelined as two stages as indicated by the dashed line in Fig. 9. Again, we need to exam the critical path of each stage. The critical paths of the first stage (left side of the dashed line in Fig. 9) and the second stage (right side of the dashed line) are expressed in Eq. (10) and Eq. (11), respectively.

$$T(stage1)_{2-step\_pre\_T} = T_{adder} + 2T_{4-in\_comp} \quad (10)$$

$$T(stage2)_{2-step\_pre\_T} = 2T_{adder} + T_{4-in\_comp} + 2T_{2-in\_comp}. \quad (11)$$

Comparing with Eq. (5), the shortest path needed for *T*-algorithm, we find that $T(stage1)_{2\text{-}step\_pre\_T}$ is shorter since the gate delay of the 2-stage 2-input comparator is slightly longer than that of a fully parallel 4-input comparator. On the other hand, $T(stage2)_{2\text{-}step\_pre\_T}$ is longer than $T_{SPEC\text{-}T\_alg}$ by the delay of an adder. However, by re-arranging the process and introducing one redundant adder, we can further reduce $T(stage2)_{2\text{-}step\_pre\_T}$ to that in Eq. (5). The details are shown in Fig. 10.

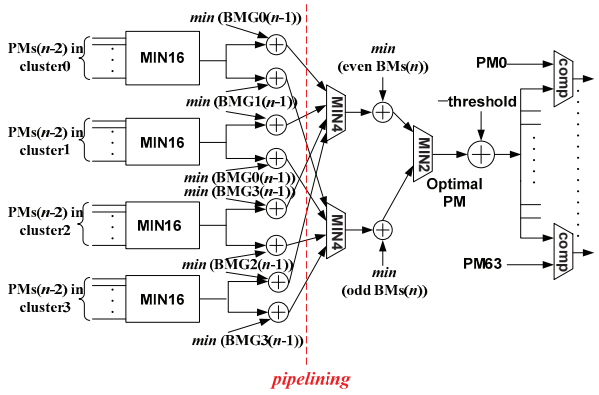

Figure 9. Implementation of 2-step pre-computation *T*-algorithm.

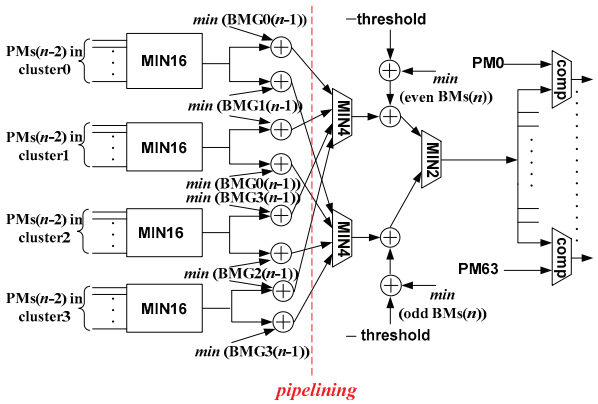

Figure 10. 2-step pre-computation *T*-algorithm with redundant operations.

In Fig. 10, the left side of the dashed line remains the same. However, at the right side, the threshold value is added to both the *min* (even BMs) and *min* (odd BMs) before PM_{opt} is found out. Now, the critical path of the PPAU for the 2-step pre-computation scheme is the same as the iteration bound in Eq. (5). Compared with the conventional implementation for *T*-algorithm, the hardware overhead of the architecture in Fig.10 includes 11 adders, a 4-input comparator and a 2-input comparator, which is about the same size of the ACS circuitry for one state.

More steps of pre-computation can be further achieved with larger hardware overhead; however, it is generally unnecessary in practice.

IV. Simulation and Implementation Results

The proposed pre-computation scheme has the same performance as the original *T*-algorithm shown in Fig.3 since for both case the accurate optimal PM is found each cycle.

In [6], it is suggested that the SPEC-*T* algorithm adjusts the estimated value every 7 cycles. However, our simulation shows that if the estimated value is adjusted every 3 or more cycles, there is a high probability (>3% at a BER $\sim= 10^{-3}$) that the decoder will lose all the survival paths during the decoding process due to the purging scheme according to the threshold and the estimated PM_{opt}. When the adjustment frequency reduces to once every 2 cycles, the probability drops down to 0.1%, which is still not acceptable for practical systems. Therefore, the SPEC-*T* algorithm must adjust the estimated value each cycle, which is equivalent to the conventional *T*-algorithm.

For a more detailed comparison, we implemented, in FPGA, the ACSU using several different schemes: 1) conventional implementation of *T*-algorithm, 2) the proposed 1-step pre-computation scheme, and 3) 2-step pre-computation schemes. SPEC-*T* algorithm is not considered here since it degrades to conventional *T*-algorithm. The synthesis results are summarized in Table I.

TABLE I. Synthesis Results

Xc4vlx160-12ff1148 Virtex 4, speed scale 12			
	Number of 4 input LUTs	Number of flip flops	Minimum period (ns)
Regular *T*-algorithm	23326(17%)	576(0%)	39.70
1-step pre-comp	21570(15%)	648(0%)	61.50
2-step pre-comp	22258(16%)	686(0%)	81.97

Table I shows that by applying the 2-step pre-computation architecture, the clock speed is doubled comparing with the conventional implementation of *T*-algorithm. It is also observed that, the pre-computation architecture requires so small hardware overhead that it is not evident in FPGA synthesis result.

V. Conclusion

In this paper, a pre-computation scheme with associated hardware architecture is proposed for Viterbi decoders employing *T*-algorithm. Compared with existing schemes that target at efficient implementation of *T*-algorithm, the proposed approach is more reliable in general. The analysis of the critical path reveals that the pre-computation scheme can achieve the iteration bound for Viterbi decoders employing *T*-algorithm with negligible hardware overhead. Simulation results show that the proposed scheme maintains the same BER performance as the conventional *T*-algorithm while other schemes could completely fail decoding. Synthesis results with FPGA have verified the significant speedup of the proposed design.

References

[1] G. D. Jorney Jr., "The Viterbi algorithm," *Proc. IEEE*, vol. 61, pp.268-278, Mar. 1973

[2] S. J. Simmons, "Breadth-first Trellis Decoding with Adaptive Effort," *IEEE Trans.Commun.*, vol. 38, no. 1, pp. 3-12, Jan. 1990.

[3] Francois Chan and David Haccoun, "Adaptive Viterbi Decoding of Convolutional Codes over Memoryless Channels," *IEEE Trans. Commun.,* vol. 45, no. 11, pp. 1389-1400, Nov. 1997.

[4] M.-H. Chan, W.-T. Lee, M.-C. Lin and L.-G. Chen, "IC design of an adaptive Viterbi decoder," *IEEE Trans. Consum. Electron*., vol. 42, pp. 52-62, Feb. 1996.

[5] Fei Sun and Tong Zhang, "Low Power State-Parallel Relaxed Adaptive Viterbi Decoder Design and Implementation," *IEEE ISCAS*, pp. 4811-4814, May, 2006.

[6] Fei Sun and Tong Zhang, "Parallel High-Throughput Limited Search Trellis Decoder VLSI Design," *IEEE Trans. Very large Scale Intergr.(VLSI) Syst.*, vol.13, no.9, pp. 1013-1202, Sept., 2005.

[7] "Bandwidth-Efficient Modulations", CCSDS 401(3.3.6) Green Book, April 2003.