BOOSTING THE IMPLEMENTATION EFFICIENCY OF VITERBI DECODERS BY NOVEL SCHEDULING SCHEMES

Stefan Bitterlich, Herbert Dawid, Heinrich Meyr
Aachen University of Technology, 5240 ERT
Templergraben 55, W–5100 Aachen, Germany

Abstract

The Viterbi algorithm is today widely used for the decoding of convolutional codes. Especially for codes achieving large high coding gains the complexity of VLSI realizations is very high. We present a novel scheduling scheme that allows to dramatically cut down the area needed to implement the ACS (Add Compare Select) Unit being the principal area consumer in high-speed Viterbi decoders (VDs). Since computation speed is only slightly compromized, a boost in implementation efficiency over existing methods can be achieved.

Since a systematic derivation for a very efficient scheduling scheme is presented the proposed scheme can be used to construct good schedules for traditional node serial processors as well. Therefore, the novel scheme is also well suited for VDs with a very high number of states.

With the proposed scheduling scheme the same hardware structure can be used to efficiently decode a set of *different codes* that may even have *different constraint lengths.* Therefore, the scheme allows for easy and efficient implementation of *programable*, high-speed VDs as well.

Furthermore, we show that the novel scheduling scheme applied to a real world Viterbi decoder, the 64 states industry standard rate 1/2, k=7 VD (CCSDS 101.0-B-2 [1]), leads to an increase of implementation efficiency of up to 600%.

1 Introduction

The growing demand of communication systems for efficient forward error correction schemes (FEC) operating at high data rates recently led to much interest in the optimal decoding algorithm, the *Viterbi Algorithm* (VA).

The VA was introduced in 1967 by Viterbi [2] as a decoding algorithm for convolutional codes. Omura [3] showed in 1969 that the VA applies the dynamic programming technique to finding the shortest path through a weighted graph. Later, Forney [4] showed that the VA is a maximum likelihood decoding algorithm for convolutional codes. A comprehensive description of the VA is also given in [4].

The implementation of the VA consists of three parts: branch metric computation, path metric updating, and survivor sequence generation. The path metric computation unit computes a number of recursive equations. In a Viterbi decoder (VD) for a N-state convolutional code, N recursive equations are computed at each time step ($N = 2^{k-1}$, k = constraint length). Existing high-speed architectures use one processor per recursion equation. The main drawback of these Viterbi Decoders is that they are very expensive in terms of chip area. In current implementations, at least a single chip is dedicated to the hardware realisation of the Viterbi decoding algorithm (for $k \geq 7$). The novel scheduling scheme allows cutting back chip area dramatically with almost no loss in computation speed.

Recently, a heuristic approach to scheduling for area-efficient VDs has been proposed [5]. The authors divide each macrocycle into a number of subcycles and then derive a scheduling scheme for the sequence of computations. The same schedule is repeated for each macrocycle. They apply their *more general* scheme to a convolutional code as one example and achieve 50% of the speed of the parallel solution with area savings of 60 %. Therefore, they increase efficiency for this case by a factor of 1.2. While the trellis is periodic with a period of exactly one macrocycle, this, at a first sight, seems to be the best suited (and obviously most natural) choice for the repetition period of the scheduling sequence.

Rather surprisingly, we found that much more efficiency can be gained in the case of *convolutional codes* by choosing a *schedule repetition period of* $\log_2 N$ macrocycles and introducing the concept of *pipeline-interleaving.* The reason for this will become clear in the sequel.

We will show that the *additional degree of freedom* for choosing a schedule gained by increasing the repetition period of the scheduling sequence leads to far more efficient solutions.

2 The ACS Recursion

The heart of the Viterbi decoder is the *add compare select (ACS) unit.* It computes the ACS recursion that essentially compares the different competing paths through the trellis and selects the maximum likelihood paths.

Such a trellis (fig.1) is described by a finite number N of states S_i at each discrete time instant k and by branches (transitions) connecting states at time instant k with states at time instant $k+1$. Each state S_i at time instant k is associated with a path-metric $\Gamma_{i,k}$, and each branch bears a weight (branch-metric) $\lambda_{ij,k}$.

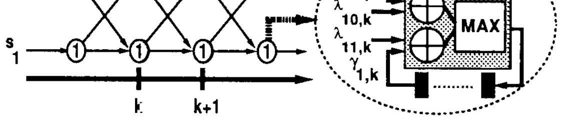


Figure 1: Trellis with N=2 states and ACS processing element (PE).

The metric $\Gamma_{i,k}$ is then updated according to the ACS recursion:

$$\forall s_i : \quad \Gamma_{i,k+1} = \underbrace{\max}_{\forall j \to i}(\lambda_{ij,k} + \Gamma_{j,k})$$

This process corresponds to the selection of most likely paths through the trellis and eliminating the unlikely (inferior) paths.

Since the (nonlinear) maximum selection is the only recursive part in the VA, the achievable data (and clock) rate is limited by the computation time of the ACS-recursion. As the ACS unit is also the main area consumer in high speed Viterbi decoders, an efficient implementation of the ACS unit is crucial for a highly efficient VD. We therefore limit our attention in this paper to the efficient implementation of the ACS unit [1].

Note that path-metric $\Gamma_{i,k+1}$ as shown in this equation can depend on all metrics $\Gamma_{j,k}$. Therefore, straightforward implementation of pipeline interleaving (discussed in the following section) is not possible. In the sequel we will show that the underlying structure of convolutional codes (that determines the specific choises for $\lambda_{ij,k}$) can be exploited to decouple these recursive equations.

3 Pipeline Interleaving

Pipeline interleaving is a well known, efficient method to increase throughput drastically with only small increases in implementation area. Adding registers in the middle of the computational paths produces a computational engine that has nearly twice (minus the time to traverse the added register) the throughput while increasing overall circuit area only by the added register's area. Thus pipeline interleaving is attractive if the additional area introduced by the registers is small as compared to the area needed to implement the computation. As computational latency is not reduced, the method is only applicable to independent calculations. Tightly coupled recursive computations where the computation at time t_n depends on the result of the preceding computation at time t_{n-1} don't benefit in any way by introducing pipeline interleaving.

4 Convolutional Codes - shuffle exchange

The trellis structure of convolutional codes resembles the interconnection scheme of shuffle-exchange (SE) networks [6]. Many different notations exist to describe the same or closely related topologies, as e.g. the cube-connected cycle (CCC), de Brujin graph, Omega graph and butterfly-graph. Throughout this paper we will adopt the following definition of a shuffle exchange graph:

- A shuffle-exchange graph is a state diagram of an L-stage shift register with an input of alphabet size B. $N = B^L$ nodes are interconnected [2].

Each node is connected to B nodes corresponding to the B possible input values of the shift register. Therefore, node

$$i = \sum_{k=0}^{L-1} B^k i_k =: (i_{L-1}, \cdots, i_0)$$

is connected to the B nodes

$$\{(i_{L-2}, \cdots, i_0, b), b \in \{0, \cdots, B-1\}\}$$

An example of an SE for $N = 2^4$ is given in fig.2.

5 TPI

The key idea of the *trellis pipeline-interleaving (TPI)* scheduling scheme is to find computational sequences in the trellis of convolutional codes that are independent from each other for as long as possible. While the sequences are independent one may use pipeline interleaved processors to compute them in a very area efficient manner. During the time those sequences merge, i.e. are dependent, we simply wait until the operands needed appear at the end of the pipeline. For example, in figure 2 we can identify four sequences (represented

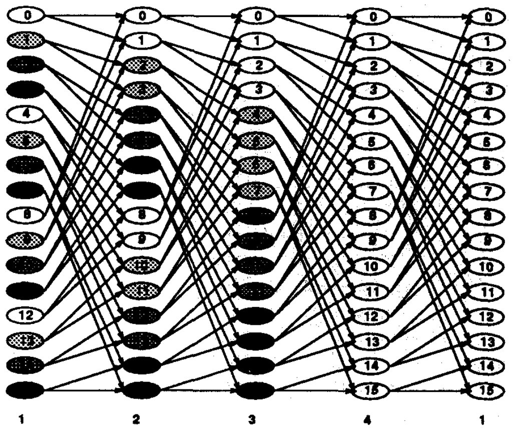


Figure 2: Shuffle-exchange graph for $B = 2, N = 16, \kappa = 2$

by different gray levels) that are independent for the first two cycles and that merge during the last two cycles.

Therefore, introducing TPI consists of three steps:

- 1. Introduce additional pipeline registers into the original ACS unit (total of B^κ pipeline registers) The TPI processing elements can be clocked at a B^κ times higher rate.

- 2. Explore the underlying structure of convolutional codes to find loosely coupled parts (where the computation can proceed independently for most of the time).

- 3. find a schedule that respects the high latency (in terms of clock cycles) while not introducing too many wait states.

In a Viterbi decoder for convolutional codes each state metric $\Gamma_{k,i}$ at time step k is computed as a function of B state metrics $\Gamma_{k-1,j}$ of the previous time step $k-1$. Represented in a base B number system, the previous state numbers are just given by a right-shift (shift towards least significant digit) of the state number inserting a new digit from the digit set $\{0, 1, \cdots, B-1\}$ into the most significant digit position:

$$\Gamma_{(i_{L-1}\cdots i_0),k+1} = f(\Gamma_{(0 i_{L-1}\cdots i_1),k}, \cdots, \Gamma_{((B-1) i_{L-1}\cdots i_1),k}) \quad (1)$$

The node parallel architecture assigns a separate processor to each equation using a total of N processors. Using our novel TPI scheduling approch we compute the N equations (1) with a set of only $N' = N/B^\kappa$ pipeline-interleaved processors. We divide each major cycle k (corresponding to the transition of time step k to time step $k+1$ in the trellis) into B^κ minor cycles and introduce B^κ pipeline stages into the traditional ACS processor. Each of these new pipeline-interleaved processors then has a throughput of one computation per minor cycle and a latency of B^κ minor cycles.

Since N values have to be computed within each major time step k, we obviously need at least B^κ (minor) time steps (one major time step) to compute these values with our processor bank.

Note that we have the *additional constraint* that the result of the computation appears only after B^κ minor cycles (= one major cycle) at the output of the pipeline-interleaved

[1]The other main area consumer is the so called survivor memory unit.

[2]This is a definition used in most VD literature. Note that the original computer-science literature definition is different.

processor. Therefore, we have to be very careful to schedule our operations in such a way that the result of a computation is never needed earlier. Preferably we try to fill up every minor cycle with a useful computation. If this is impossible, we will have to insert some wait states arriving at more than B^κ minor cycles for the complete computation.

We will now present a scheduling scheme that introduces only an astonishingly small number of wait states. This allows to gain nearly the complete potential of the standard pipeline-interleaving technique as used for independent computations.

5.1 Micro scheduling ACS computations

In describing the TPI scheduling scheme we make heavy use of the base B digit representation of certain quantities. In this paper, we write the base B representation of quantity x as $(x_{n-1}\cdots x_0)$ with the convenient meaning $x = \sum_{i=0}^{n-1} x_i B^i$.

Let m denote the minor scheduling cycle where $0 \le m \le B^\kappa - 1$ and $(m_{\kappa-1}m_{\kappa-2}\cdots m_0)$ the base B digit representation of the number m.

Further, let $p = \sum_{i=0}^{L-\kappa-1} B^i p_{i+\kappa} =: (p_{L-1}p_{L-2}\cdots p_\kappa)$ be the processor number the computation is assigned to.

The TPI scheduling uses a scheduling period of $\log_2 N$ major cycles (corresponding to $\log_2 N$ time steps of the trellis) instead of just one major cycle. The micro scheduling which is different for each of the $\log_2 N$ major cycles, is given in table 1.

Note that the scheduling table is written in a somewhat condensed way. Every line in the table gives the minor scheduling and processor allocation for one specific major cycle. It specifies which recursion equation is computed in which minor cycle on which processor. Each entry represents the set of the N recursion equations. For each recursion equation number (represented digitwise as a base B number) it specifies the (digitwise) decomposition into processor number and minor schedule number. For example, in the case $B = 10, L = 5, \kappa = 3$ and recursion equation no.12345 in major cycle 1 we associate $p_{L-1} = 1, p_{L-2} = 2$. Therefore processor 12 is used. $m_0 = 3, m_1 = 4, m_2 = 5$ indicates that this equation is scheduled for computation in minor phase 543 (note that digits appear reversed because m_0 is the least significant digit of the minor schedule cycle number).

major cycle no.	minor schedule	minor waits after cycle
1	$(p_{L-1}p_{L-2}\cdots p_\kappa m_0\cdots m_{\kappa-1})$	
2	$(p_{L-2}p_{L-3}\cdots p_\kappa m_0\cdots m_{\kappa-1}p_{L-1})$	
$\vdots$		
α	$(p_{L-\alpha}\cdots p_\kappa m_0\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\alpha+1})$	
$\vdots$		
L-κ	$(p_\kappa m_0\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa-1})$	
L-κ+1	$(m_0\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa})$	B-1
L-κ+2	$(m_1\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}m_0)$	$(B$-1$)B$
L-κ+3	$(m_2\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}m_0m_1)$	$(B$-1$)B^2$
L-κ+4	$(m_3\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}m_0m_1m_2)$	$(B$-1$)B^3$
$\vdots$		
L		$(B$-1$)B^{\kappa-1}$

Table 1: general scheduling table for TPI

To clarify the notation used we give a textual interpretation of the entry of major cycle 1 in the case $B = 2, L = 4$ and $\kappa = 2$ (the original trellis is given in fig.2). A graphical representation of this scheduling is shown in figure 3. The scheduling table expanded by the minor states is given in figure 4. The binary digits a,b are arbitrary. They give the processor number which the corresponding recursive equation is assigned to (e.g.in major cycle 4 minor cycle 0 entry 0ba0: processor no.00 computes equation $0\underline{00}0$, processor no.01 computes equation $0\underline{01}0$, processor number 10 computes eq. $0\underline{10}0$ and processor number 11 computes eq. $0\underline{11}0$).

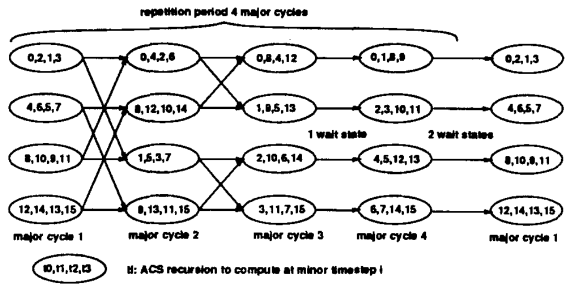
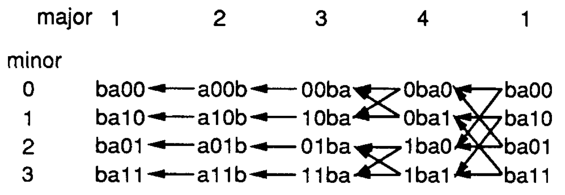


Figure 3: TPI scheduling for N=16 nodes, $\kappa = 2$



major 1 2 3 4 1
minor
0 ba00 a00b 00ba 0ba0 ba00
1 ba10 a10b 10ba 0ba1 ba10
2 ba01 a01b 01ba 1ba0 ba01
3 ba11 a11b 11ba 1ba1 ba11

Figure 4: TPI scheduling for N=16 nodes, $\kappa = 2$

The scheduling entry (major cycle 1)

$$p_{L-1}p_{L-2}\cdots p_\kappa m_0\cdots m_{\kappa-1}$$

means that here the first $L - \kappa = 2$ digits of the recursion equation number represent the processor the recursion is calculated on. The last κ digits give the minor scheduling cycle during which the specified recursion is calculated. Therefore, in minor cycle 0 (corresponding to $m_1 m_0 = 00$) we have to compute ACS recursion number (0000) on processor 0 (corresponding to $p_{L-1}p_{L-2} = 00$). Concurrently the recursions $(\underline{01}00)$, $(\underline{10}00)$, $(\underline{11}00)$ are calculated on processors 1 to 3. In minor cycle 1 $(m_{\kappa-1}\cdots m_0 = 01)$ recursions $(\underline{00}10)$, $(\underline{01}10)$, $(\underline{10}10)$, $(\underline{11}10)$ are computed concurrently on processors 0 to 3, respectively (the digits of the recursion equations that give the processor numbers are underlined).

The arrows in fig.4 indicate the computational dependencies that force the introduction of 1 and 2 minor wait states following major cycles 3 and 4. This is necessary because results originating from the *same* processor but from different minor cycles of the previos major cycle are to be combined (e.g. all equations of the form 0ba0 in major cycle 4 minor cycle 0 need the results of the computation in major cycle 3 minor cycles 0 and 1 of the equations of the form 10ba with b,a arbitrary. This is because the left shift of 10ba is 0ba0 indicating a direct dependency.).

In this example $2^2 = 4$ minor cycles with each cycle concurrently computing $2^{4-2} = 4$ ACS recursions are used to compute the $2^L = 16$ recursion equations of one time step in the trellis. Note that every recursion equation number occurs exactly once during each major cycle. Therefore, the computations of exactly one trellis step are completed during the major cycle.

5.2 Computational efficiency

We will now analyze the presented scheduling scheme to compute the number of wait states needed to fullfil the constraint of at least B^κ computation cycles that is enforced by the pipeline-interleaved processors. Then we will count the total number of wait states required and compute the overall implementation efficiency of the scheme.

The computational dependencies in an ACS for convolutional codes are given by the set of equations (1). Now according to (1) the nodes

$$\{p_{L-2}p_{L-3}\cdots p_\kappa m_0 \cdots m_{\kappa-1}p_{L-1}, p_i \in [0,\cdots,B-1]\}$$

of major cycle 2 depend on the set of nodes

$$\{xp_{L-2}p_{L-3}\cdots p_\kappa m_0 \cdots m_{\kappa-1},$$
$$x \in [0,\cdots,B-1], p_i \in [0,\cdots,B-1], i \neq L-1\}$$

(just right-shifting and inserting an arbitrary digit for the most significant digit, see section 4). But these are the nodes computed in major cycle 1 within the *same* minor cycle. Thus there are 2^κ minor cycles between these interdependent calculations, and this exactly is what is enforced by the 2^κ pipeline stages of the pipeline interleaved processors. Therefore, here we need no wait state at all. Analogous argumentation for major cycles 2 to $L-\kappa+1$ is possible. Up to major cycle $L-\kappa+1$ no wait states are needed.

The first problem occors at major cycle $L-\kappa+2$. The set of nodes $\{(m_1\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}m_0), p_i \in [0,\cdots,B-1]\}$ depends on the results of the set of nodes

$$\{(xm_1\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}), x \in [0,\cdots,B-1],$$
$$p_i \in [0,\cdots,B-1]\}$$

Because x now fills an m_0 instead of an p_i position of the previous major cycle $L-\kappa+1$, the set needed is not available at the same time (same m_i but different p_i means that something is computed in the same minor cycle, but on a different processor. But different m_i means the computation takes place during a different minor cycle.) Merely the nodes

$$\{(xm_1\cdots m_{\kappa-1}p_{L-1}\cdots p_{L-\kappa}), p_i \in [0,\cdots,B-1]\}$$

are computed in minor cycle $(m_\kappa\cdots m_1 x) = m + x$. The worst possible case (latest computation) is given by $x = B-1$. Therefore, we have to insert $B-1$ wait states (each of the duration of one minor cycle) after major cycle $L-\kappa+1$.

An analogous argument can be given for the major cycles $L-\kappa+2$ to L. Here not just the least significant digit but higher order digits of the minor scheduling cycle are different. In general, after major cycle $L-\kappa+i$ a number of $(B-1)B^{i-1}$ wait states are needed ($i \in [1,\cdots,\kappa]$).

Thus the total number of wait states, each of the duration of a minor cycle is:[3] $W = \sum_{i=1}^{\kappa}(B-1)B^{i-1} = B^\kappa - 1$.

The total number of minor cycles for computing the $\log_2 N$ major cycles consists of LB^κ minor computing cycles and W wait states which amounts to $LB^\kappa + W = (L+1)B^\kappa - 1$.

This results in a relative time degradion compared to the node parallel approch of $T/T_0 = 1 + \frac{1}{L}(1 - B^{-\kappa})$.

Therefore, using reasonable values of L, TPI experiences only a slight degradation of the overall computation time while consuming only $1/B^\kappa$ times the area (Of course, the scheme doesn't make sense if the original trellis has only one or two states per time step in the trellis). The *overall 1/AT efficiency* η thus is improved by a factor of

$$\frac{\eta_{TPI}}{\eta_0} = \frac{B^\kappa}{1 + \frac{1}{L}(1 - B^{-\kappa})}$$

over the efficiency of the fully parallel scheme.

As shown in [7] the communication structure of the $N' = N/2^\kappa$ pipeline-interleaved processors *again forms a shuffle exchange interconnection network* only of smaller size N'. In

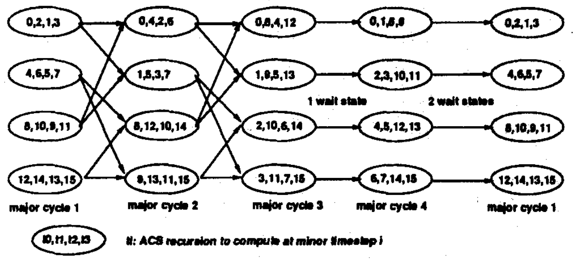


Figure 5: TPI scheduling for N=16 nodes, $\kappa = 2$, inner nodes of major cycle 2 exchanged

the example, interchanging the inner nodes in major cycle 2 leads to the SE structure (fig.5).

Major cycles where wait states can be avoided map onto a shuffle exchange, while the other cycles map onto local communication *within* the processors (every processor needs only previous results from itself). Therefore, no complicated storage management is needed. The storage system can be implemented very efficiently by simple *shift registers*.

5.3 Communication efficiency

The shuffle exchange interconnection network is a *global* communication scheme. It has been proven that the area complexity of a shuffle exchange ($B = 2$) is of $\Omega(N^2/\log^2 N)$ [8,9]. Thus communication becomes a severe problem when implementing high speed node parallel Viterbi decoders. The TPI scheduling scheme reduces the number of node processors by a factor of 2^κ. These $N/2^\kappa$ node processors are connected again by a smaller shuffle-exchange network. Therefore, the area needed for communication is reduced by a factor of

$$\frac{A_{0,cm}}{A_{TPI,cm}} = \frac{N^2/\log^2 N}{(\frac{N}{2^\kappa})^2/\log^2(\frac{N}{2^\kappa})} = 2^{2\kappa}\frac{(\log(N)-\kappa)^2}{\log^2 N}$$

Furthermore, CMOS circuit switching frequencies *depend heavily* on the capacitive load of gate outputs (usually a linear dependency between capacitive load and switching time is assumed). Wiring capacitance depends linearly on the wiring area. Therefore, the communication time (time until the signal reaches the end of the wire) is reduced by the factor $2^{2\kappa}\frac{(\log(N)-\kappa)^2}{\log^2 N}$, too. The pipeline-interleaved processor bank must operate at 2^κ times the clock rate of the original structure. Each of the $N/2^\kappa$ processors has to send 2^κ results over the smaller SE network during the time each of the original N processors used to send one. Concluding, the overall gain in computation speed for the TPI scheme reduces to

$$\frac{T_{0,cm}}{T_{TPI,cm}} = 2^\kappa\frac{(\log(N)-\kappa)^2}{\log^2 N}$$

The reduced communication time can be used to *compensate* the loss in computation speed due to the inserted wait-states.

Hence, the overall $1/AT$ communication efficiency η_{cm} is increased by a factor of

$$\frac{\eta_{TPI,cm}}{\eta_{0,cm}} = 2^{3\kappa}(1 - \frac{\kappa}{\log N})^4$$

We can therefore conclude that in addition to the higher computation efficiency the TPI scheme has significant advantages in communication area and time, i.e. *communication efficiency*, as well.

[3]using the formula $\sum_{i=1}^{n} q^{i-1} = \frac{q^n-1}{q-1}$ for the summation of a geometric series

5.4 Flexibility

The TPI scheduling scheme can be combined with existing architectures for ACSUs of VDs (wordlength w):

- carry ripple scheme for ACS operation ($w+2$ full-adders in critical path)
- carry look-ahead scheme for ACS operation ($2 \log w$ gate delays in critical path)
- carry save scheme (2 full adder plus 2 carry save maximizer delays in critical path) [10]

These existing architectures for the ACS node processor all have a relatively long critical path (typically 20 to 40ns in 1.6μ CMOS). Therefore, there is enough potential for further pipelining by applying the TPI scheduling scheme. This leads to a class of new combined architectures that share the common feature of a highly efficient ACS unit. Thus, the designer is free to choose a particular architecture due to the specific needs in terms of throughput, area, and power-consumption and combine it with TPI.

If, for example, the throughput of standard word-level ripple-carry or carry-look-ahead schemes for adder and comparator in the ACS unit is too low, one can combine TPI with the recently proposed carry-save ACSU schemes [10]. This produces an architecture with a much higher efficiency than the original CS architecture while maintaining its high speed.

6 Application to industry standard 64 States Viterbi Decoder

To demonstrate the significance of the proposed scheme we show the application of TPI scheduling to the industry standard 64 states constraint length 7, rate 1/2 VD (CCSDS 101.0-B-2 [1]).

Current implementations achieve clock frequencies of 25 MHz with $1.5\mu m$ CMOS technologies. Therefore, the critical path is quite long ($40ns$). By applying TPI scheduling, considerable efficiency improvement is possible, even by ***semi-custom*** design. As semi-custom designs with clock frequencies of 50..100MHz are possible one ($\kappa = 1$) or three ($\kappa = 2$) additional pipeline stages can be inserted. This leads to an efficiency improvement of 85% ($\kappa = 1$) or 250% ($\kappa = 2$) [4].

κ	$\frac{A_{0,cm}}{A_{TPI,cm}}$	$\frac{\eta_{tpi,cm} - \eta_{0,cm}}{\eta_{0,cm}} * 100\%$	$\frac{\eta_{tpi,cp} - \eta_{0,cp}}{\eta_{0,cp}} * 100\%$
1	2.8	285 %	85 %
2	7.1	1160 %	256 %
3	16.0	3100 %	598 %

Table 2: Improvement of communication area A_{cm}, efficiency η_{cm} and computation efficiency η_{cp} for $N = 64$.

The full potential of TPI can be exploited by ***full-custom*** design. The recently proposed ***single phase clocking*** schemes allow very high clock frequencies (multiples of 100 MHz) even with $1.6\mu m$ CMOS technologies [5]. Therefore, it can be expected that implementations up to $\kappa = 3$ are possible by hand-layout thereby reducing the critical path to about 5ns. This corresponds to the introduction of 7 new pipeline stages into the ACS unit. 600% improvement in computation efficiency is possible for $\kappa = 3$. In this case the communication area is reduced to 1/16 of the traditional node parallel solution, and the communication efficiency increases by a factor of 32.

[4]neglecting implementation area of registers

[5]It is rather surprising that circuits with such high operating frequencies are possible in $1.6\mu m$ CMOS technology. These single phase clocking schemes appeared only recently (1987). Yuan et al. report a 650MHz accumulator using $1.6\mu m$ n-well CMOS technology [11].

7 Conclusions

In this paper we derive a novel scheduling scheme (TPI) that allows for efficient implementation of high speed Viterbi decoders for convolutional codes. We exploit the properties of the underlying structure of convolutional codes to calculate the overall efficiency improvement that can be achieved applying TPI. Furthermore, we show that not only computation efficiency is highly improved but communication efficiency as well. The TPI scheme can be combined with existing ACS architectures. Thus a class of new combined architectures can be derived.

Because the scheme is based on the properties of shuffle-exchange interconnection networks it can be applied to other SE processors (e.g. FFT, sorting, polynom evaluation) as well.

For a real world example, the industry standard 64 states constraint length 7, rate 1/2 VD (CCSDS 101.0-B-2 [1]), we demonstrated the significance of the results. Up to 600 % implementation efficiency improvement over the node parallel high speed architecture can be achieved in this case.

References

[1] "Consultative committee for space data systems: Recommendation for space data system standards, telemetry channel coding," in *CCSDS 101.0-B-2 Blue Book*, 1987.

[2] A. Viterbi, "Error bounds for convolutional coding and an asymptotically optimum decoding algorithm," *IEEE Trans. Information Theory*, vol. IT-13, pp. 260–269, April 1967.

[3] A. Omura, "On the Viterbi algorithm," *IEEE Trans. Information Theory*, pp. 177–179, January 1969.

[4] G. Forney, "The Viterbi algorithm," *Proceeding of the IEEE*, vol. 61, pp. 268–278, March 1973.

[5] C. B. Shung, H.-D. Lin, P. H. Siegel, and H. K. Thapar, "Area-efficient architectures for the Viterbi algorithm," in *IEEE Global Telecommun. Conf. (GLOBECOM)*, vol. 3, (San Diego, CA), pp. 1787–1793, December 1990.

[6] P. Gulak and E. Shwedyk, "VLSI structures for Viterbi receivers: Part I - general theory and applications," *IEEE Journal on Sel. Areas in Commun.*, vol. SAC-4, pp. 142–154, Jan. 1986.

[7] H. Dawid, S. Bitterlich, and H. Meyr, "Trellis pipeline-interleaving: A novel method for efficient Viterbi-decoder implementation," in *Proceedings IEEE ISCAS*, (San Diego), pp. 1875–78, 1992.

[8] D. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller, "An asymptotically optimal layout for the shuffle-exchange graph," *Journal of Computer and System Sciences*, vol. 26, pp. 339–361, 1983.

[9] F. P. Preparata and J. Vuillemin, "The cube-connected cycles: A versatile network for parallel communication," *Communications of the ACM*, vol. 24, pp. 300–309, May 1981.

[10] G. Fettweis and H. Meyr, "High-speed Viterbi processor: A systolic array solution," *IEEE Journal on Sel. Areas in Commun.*, vol. SAC-8, pp. 1520–1534, Oct. 1990.

[11] J. Yuan, C. Svensson, F. Lu, and H. Samueli, "A high speed pipelined CMOS accumulator for implementing numerically controlled oscillators," in *ISCAS*, (New Orleans, LA), 1990.