Survivor path processing in Viterbi decoders using register exchange and traceforward

Kamuf, Matthias; Öwall, Viktor; Anderson, John B

Published in:
IEEE Transactions on Circuits and Systems II: Express Briefs

DOI:
10.1109/TCSII.2007.891753

2007

Survivor Path Processing in Viterbi Decoders Using Register Exchange and Traceforward

Matthias Kamuf, *Student Member, IEEE*, Viktor Öwall, *Member, IEEE*, and John B. Anderson, *Fellow, IEEE*

***Abstract*—This brief proposes a new class of hybrid VLSI architectures for survivor path processing to be used in Viterbi decoders. The architecture combines the benefits of register exchange and traceforward algorithms, that is, low storage requirement and latency versus implementation efficiency. Based on a structural comparison, it becomes evident that the architecture can be efficiently applied to codes with a larger number of states where traceback-based architectures, which increase latency, are usually dominant.**

***Index Terms*—Convolutional codes, register exchange (RE), survivor path, traceback (TB), traceforward (TF), Viterbi decoder, VLSI.**

I. Introduction

THE Viterbi algorithm is a maximum-likelihood algorithm that can be applied to decoding of convolutional codes. In this brief, we consider convolutional codes of rate $1/c$, c an integer, and high-rate punctured codes that are derived from them. Their trellises have $N = 2^m$ states, where m is the encoder memory.

A Viterbi decoder typically consists of three building blocks, as in Fig. 1:

- a branch metric unit (BMU) that calculates the likelihood for the possible transitions in a trellis;
- add-compare-select units (ACSUs) that discard suboptimal trellis branches based on current branch metrics and previously accumulated state metrics;
- and a survivor path unit (SPU) that works upon the decisions from the ACSUs to produce the decoded bits along the reconstructed state sequence through the trellis.

The ACSUs and SPU are known to be critical parts in a hardware implementation. In particular, the algorithm used for the SPU affects the overall memory requirement and latency of the decoder, two important aspects in today's communication systems. SPU algorithms rely on the fact that the survivor paths are expected to have merged with sufficiently high probability after a certain decoding depth L.

Traditional approaches for the SPU are register exchange (RE) and traceback (TB) [1] algorithms. RE has the lowest memory requirement (NL bits) and latency (L) among all SPU algorithms. However, NL bits must be read and written every cycle, making an implementation in high density random access memory (RAM) impractical. Instead, the algorithm is preferably realized by a network of multiplexers and registers that are connected according to the trellis topology. For a larger number of states, though, RE results in power-hungry implementations due to the low integration density of the multiplexer-register network and the high memory bandwidth. RE is primarily used in applications where high speed and low latency are crucial design parameters.

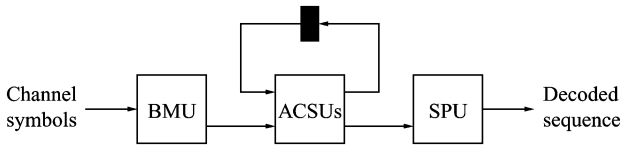

Fig. 1. Principal block diagram of a Viterbi decoder.

TB is considered applicable to an almost arbitrary number of states at the cost of an increase in both memory and latency. This method is a backward processing algorithm and requires the decisions from the ACSUs to be stored in much denser memory, typically RAM. Only N decision bits are written every cycle, thus the write access bandwidth is greatly reduced. For an n-pointer-odd architecture [1], where $n > 1$ is the number of read pointers, the RAM requirement is $NL(2 + 1/(n - 1))$. Furthermore, since information symbols are recovered time-reversed, a last-in-first-out (LIFO) buffer of size $L/(n - 1)$ has to be introduced to reverse the decoded bitstream. To reduce the increased memory size and latency, TB is mainly used in conjunction with the traceforward (TF) [2] procedure, which is discussed in Section II-B.

After a brief review of existing hybrid SPU architectures in Section II, a new hybrid approach based on RE and TF is proposed in Section III. Storage requirement and latency can be traded for implementation complexity. Therefore, this architecture can be applied to a larger number of states, which is justified by a comparison to existing hybrid approaches in Section IV.

II. Existing Hybrid Approaches

Several attempts have been made to increase the implementation efficiency of the SPU by combining different algorithms. Two prominent members are discussed in the following since these are the ones our architecture is derived from and competes with.

A. RE and TB

A hybrid architecture combining RE and TB was first published in [3]. The idea was also discovered in [2] and [4], and

Manuscript received July 25, 2006; revised October 16, 2006. This work was supported by the Competence Center for Circuit Design at Lund University. This paper was recommended by Associate Editor B. Nikolic.

M. Kamuf and V. Öwall are with the Department of Electroscience, Lund University, Lund S-221 00, Sweden (e-mail: mkf@es.lth.se; vikt@es.lth.se).

J. B. Anderson is with the Department of Information Technology, Lund University, Lund S-221 00, Sweden (e-mail: anderson@it.lth.se).

Digital Object Identifier 10.1109/TCSII.2007.891753

later generalized in [5] to derive a class of so-called pre-compiled SPUs. This class also includes the TF approach, which they call $ER(\nu, L)$ precompilation. However, except for the TF method, these approaches require specific memories which have to be accessible row- and columnwise.

To reduce both latency and read accesses during the merge phase, the RE/TB architecture carries out TB operations over D bits at a time instead of one. Let the decoding depth L be divided into blocks of size D, that is, $L = kD$, k an integer. An RE unit (REU) of size ND is used to continuously build segments of the survivor path for each state. These segments are then stored every step as N-bit *column* vectors in a RAM. A D-bit segment of a *row* vector for a certain state contains the starting state of its survivor path D bits earlier; that is, this so-called block TB covers D bits per RAM read access, instead of one bit as in the traditional TB method. Hence, the number of TB operations to find the starting state of a decoding segment is lowered from L to k. Since the survivors are preprocessed in the REU, the final decoding can be carried out in one step. Note, however, that the RAM has to be accessible both row- and columnwise, which requires a more complex specialized memory implementation. The overall storage requirement, distinguished by implementation complexity, becomes

$$\underbrace{ND}_{\text{REU}} \text{ and } \underbrace{N(L+(2p-1)D)}_{\text{RAM}} \tag{1}$$

where p is the number of D-bit segments that are finally decoded once a starting state is found [3].

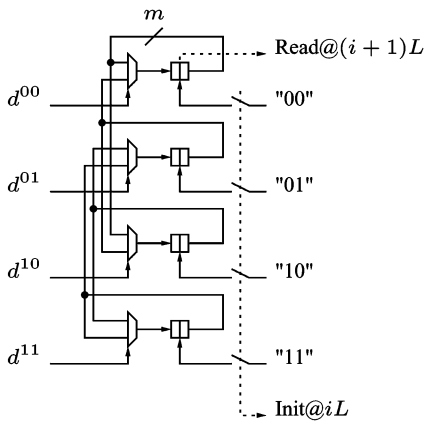

Fig. 2. TFU for a 4-state trellis, that is, $m = 2$. Figure adapted from [2].

it is also indicated in [2] that a TFU can be optimized individually, where area-efficient ACSU topologies are applied due to the structural equivalence of TFU and ACSU.

The extension to TB architectures with k TFUs that estimate starting states at times L/k to further reduce latency is discussed in [6]. In total the storage requirement is

$$\underbrace{N \times L\left(1+\frac{1}{k}\right)}_{\text{RAM}}, \quad \underbrace{kNm}_{\text{TFU}}, \text{ and } \underbrace{\frac{L}{k}}_{\text{LIFO}} \tag{2}$$

bits in this approach.

B. TF and TB

In agreement with its first appearance in the literature [2], we adopt the name TF for the following procedure. An algebraic generalization is found in [6] and real hardware effects of this approach have been recently published in [7] and [8].

TF is a forward-processing algorithm that estimates the starting state for a decode TB on-the-fly such that TB operations during a merge phase, which do not contribute to the actual decoding, can be omitted. The TF method is applied to lower both memory requirement and latency in TB-based architectures.

Every survivor path at time $i+\Delta$, i an integer, is connected to some state at time i, called a tail state. According to the lemma about decoding depth, all survivor paths should stem from the same state for $\Delta > L$, that is, their tail states are identical.

Fig. 2 shows the TF unit (TFU) for a 4-state rate $1/c$ convolutional code, that is, $N = 4$ and the encoder memory is $m = \log_2 N = 2$. At time iL, each m-bit register block is initialized with the state label it represents; that is, current states and tail states are identical, and the survivor paths are of zero length. The decision d^{xx} for state xx selects the tail state of its predecessor state to update the current tail state. At time $(i+1)L$, when all survivor paths have merged, all registers contain the starting state for the decoding segment at time iL. For illustration, state "00" is chosen to be read from in Fig. 2. Furthermore,

III. Proposed Hybrid Approach: RE and TF

As stated in [6], the starting state of decoding segments can be found by means of so-called multiplier feedback loops, which are equivalent to TFUs. According to this observation, we estimate the starting states of length-D segments with TFUs in intervals of D; see Fig. 3. Every Dth step, a TFU is initialized, and a total of k TFUs are needed to cover the complete decoding depth L; that is, TFU_j, for $j = 1, \ldots, k$, is initialized at time $(i-1)L + jD$. Then, at time $iL + jD$, TFU_j contains the estimated starting state of this segment.

Contrary to the previous hybrid approaches, the sequences in the REU are not used for initializing a block TB operation. Instead, these partial survivor sequences are stored every Dth step in a RAM with first-in-first-out (FIFO) access that can be implemented in a much denser fashion than the original RE network of length L. Once an estimated starting state is established, the respective partial information sequence is directly read from the FIFO. Therefore, time reversal upon decoding as in hybrid TB-based architectures becomes unnecessary and the latency is not increased.

The proposed SPU architecture is depicted in Fig. 4. It consists of three parts: an REU to continuously update the partial survivor sequences for each state, a FIFO to store k sets of N sequences, and a bank of k TFUs that provide the starting states of the length-D segments.

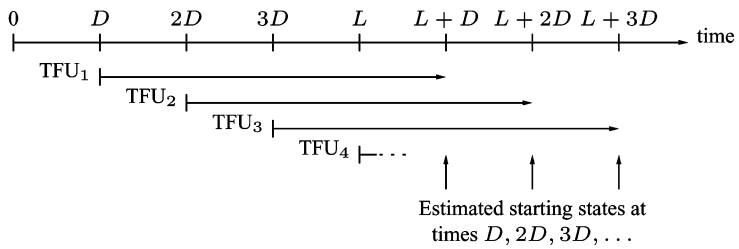

Fig. 3. Picture of TF and decode flow. Note that $k = L/D$ is an integer; in this example $k = 4$.

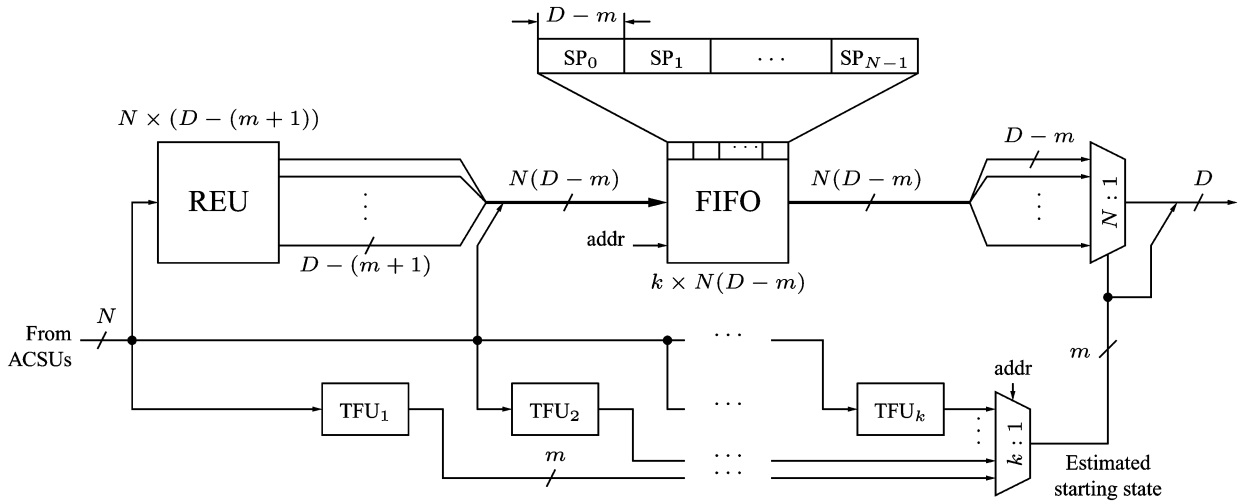

Fig. 4. Proposed hybrid SPU for feedforward codes. Shown above the FIFO is the address pattern for a partial survivor sequence word. The word consists of N sequences SP_S of length $D - m$. The FIFO could be organized for $(D - m)$-bit read accesses, that is, multiplexer $N : 1$ is incorporated in such a specialized memory.

The following considerations focus on feedforward codes, that is, an estimated starting state is equivalent to the last m information bits that entered the encoder.[1] In a straightforward implementation, an REU of length D is needed. We note, though, that for feedforward codes the start of all partial survivor sequences is always the same until the trellis is fully extended, that is, after m steps. Thus, only $D - m$ stages are required. Additionally, due to the trellis topology, the last column of decision bits can be directly transferred to the FIFO without storing them in the REU of length $D-(m+1)$. Note that there is a constraint on the minimum feasible block length $D \geq m$.

At times iD, for each state, a partial survivor sequence from the REU is stored in the FIFO which is disabled otherwise. The storage scheme of these sequences is shown above the FIFO. Here, SP_S, $S = 0 \ldots N - 1$ denotes an information stream of length $D - m$ associated with state S. It is seen that the sequence SP_S resides at address S of the memory word. To find the part that is linked to the actual survivor path, the estimated starting state from a TFU is used. For example, at time $L + D$, TFU_1 contains the starting state of the surviving path at time D and the FIFO subword at this address is selected. These bits represent the information sequence from time 0 to $D - m - 1$. The remaining m bits are included in the estimated starting state since it is equivalent to the information sequence that entered the encoding shift register. Hence, the overall latency of this approach is $L + D$. For feedback codes, these remaining bits are delivered by the REU, which now is of length $D - 1$.

Both REU and TFUs are controlled by the ACSU decisions and run continuously at data rate, whereas the FIFO only runs at $1/D$ times the data rate. The FIFO and the multiplexer $k : 1$ both use the same address counter; compared to TB architectures with multiple pointers that require independent address counters, control is much simpler. The estimated starting state selects the subword of length $D - m$ by accessing the $N : 1$ multiplexer. No reversal of the output sequence is required since only forward processing algorithms are used. This preserves low latency.

In summary, the total memory requirement becomes

$$\underbrace{N(D-(m+1))}_{\mathrm{REU}}, \quad \underbrace{k \times N(D-m)}_{\mathrm{RAM}}, \text{ and } \underbrace{kNm}_{\mathrm{TFU}}. \tag{3}$$

The architecture is scalable by varying D, thus trading memory requirement for implementation complexity. Different L require different partitions between the processing blocks (FIFO, TFU_j, REU) to optimize the implementation. Moreover, an optimal partition depends on the implementation platform. Two special cases can be pointed out for feedforward codes, namely $D = m$ and $D = m + 1$. In both cases, the REU becomes redundant. In the former case, the FIFO also vanishes and the architecture solely consists of TFUs.

[1] An information bit enters the encoder and causes a state transition, and a decision bit is put out from an ACSU upon decoding. The latter thus indicates a surviving branch in the trellis diagram. Note that information and decision bits for a state are not the same for feedforward codes but coincide for feedback codes.

TABLE I
KEY PERFORMANCE FEATURES OF DIFFERENT SPU ARCHITECTURES. STORAGE REQUIREMENT FROM (1)–(3) WAS REFORMULATED TO ALLOW FOR EASIER COMPARISON

	REU	RAM	TFU	LIFO	Latency
RE	NL	—	—	—	L
TB[a]	—	$NL(2+1/(n-1))$	—	$L/(n-1)$	$L(2+2/(n-1))$
RE/TB [3][b]	ND	$NL(1+(2p-1)/k)$	—	—	$L(1+p/k)$
TB/TF [2],[7],[8]	—	$NL(1+1/k)$	kNm	L/k	$L(1+2/k)$
Proposed (RE/TF)	$N(D-(m+1))$	$NL(1-m/D)$	kNm	–	$L(1+1/k)$

[a]The number of read pointers $n = k + 1$.
[b]REU requires ND additional multiplexers. RAM must be accessible row- and columnwise and should be organized for D-bit read accesses.

IV. COMPARISON AND DISCUSSION

Table I lists SPU architectures and their key performance features to allow for comparison between different methods. These comparisons are concerned with the hybrid approaches only, RE and TB algorithms are mentioned for completeness.

Considering the RE/TB method from [3], it is seen that it lags our approach when it comes to RAM requirements. More specifically, given the same latency $(p = 1)$ their RAM size is larger by

$$N(km + D) \text{ bits.}$$

Comparing the number of register bits, their REU has $m + 1$ extra stages. However, due to k TFUs in our approach, there are now an additional $N((k-1)m-1)$ register bits compared to [3]. Note that this is not necessarily the only measure for RE complexity. For example, a TF operation can be executed sequentially in $l \leq m$ steps, which lowers the numbers of multiplexers and interconnections by a factor l. This observation concurs with [6], where the complexity of a multiplier feedback loop $(\equiv \mathrm{TFU})$ is that of one stage of RE since they both operate sequentially on one single decision matrix at a time. Therefore, the complexity of additional k TFUs are comparable to k stages of RE network. That is, the D RE stages in [3] can be set into relation to $D-(m+1)+k$ equivalent RE stages in our approach. The RE complexity in our architecture is reduced if

$$D > D-(m+1)+k. \tag{4}$$

Since the decoding depth can be expressed as a multiple (ρ) of the code constraint length, that is, $L = \rho(m+1)$, (4) holds if $D > \rho$.

Apart from that, the REU from [3] has to employ a so-called "zone limit," which distinguishes between RE mode and shift register mode. This increases implementation complexity due to a multiplexer in front of every register in the REU, which requires ND additional multiplexers.

Another drawback in [3], which also applies to almost all of the pre-compiled approaches from [5], is that the RAMs have to be accessible row- and columnwise, which requires a specialized memory implementation and increases complexity. Decision bits are written on a per-state basis (columns) and are read on a per-time-instance basis (rows). This requirement could possibly be dropped by means of a pre-buffering scheme to do the required transposition, which on the other hand increases register complexity.

The comparison to TB/TF is carried out on the basis of same latency. From Table I it is seen that there are twice as many pointers k needed in the TB/TF approach compared to our method. Let k_1 and k_2 denote the number of TFUs in RE/TF and TB/TF methods, respectively, and $k_2 = 2k_1$. Now the different units can be compared in terms of complexity. To start with the RAM, k_2 partitions are necessary in TB/TF, which increases peripheral overhead. Note that this overhead is not considered in the following calculations. On the contrary, our method needs only one single RAM block, independent of k_1. Simplifying the difference of the RAM sizes, it is seen that TB/TF requires an additional

$$N\left(k_1 m + \frac{1}{k_2}\right) \text{ bits.}$$

Since there are twice as many TFUs, there are $k_1 Nm$ additional bits in TB/TF. Since the number of bits is comparable to the ones in the REU, we can directly subtract this overhead from the size of our REU. Furthermore, because of the small size of the LIFO ($D_2 = L/k_2$ bits), an implementation with registers is favourable compared to RAM cells. Based on these observations, the register overhead becomes

$$N(D_1 - [1 + m(k_1+1)]) - D_2 \text{ bits} \tag{5}$$

in our approach. Equation (5) grows with $\mathcal{O}(N)$ and depending on the sign of the expression in the parenthesis this overhead is in or against our favour. If $1 + m(k_1+1) > D_1$, the register complexity in our approach is smaller. Modifying this inequality gives

$$(k_1 - \rho)(m+1) + mk_1^2 > 0$$

which holds for many parameter choices apart from the obvious $k_1 \geq \rho$. It is clear that k and N are critical parameters when comparing implementation efficiency of RE/TF and TB/TF. One should also keep in mind that the complexity of a TFU compared to an REU can be adjusted in many ways as mentioned earlier.

Generally, the different architectures' feasibility depend on the choice of implementation parameters. That is, a factor sets

the cost of register and RAM bits into relation. Such a factor would depend on the number of RAM bits, partitions, RE interconnects, folding of TF operations, and so on.

The proposed architecture is seen as means to lower the RE complexity by employing denser storage cells for the survivor sequences. Thus, the architecture can be applied to codes with larger number of states. At the same time, the desirable high-speed low-latency feature of RE is preserved.

Throughout the preceding considerations, we assumed a two-port memory for the FIFO that allows a read-before-write access on the same address, so the old value is present at the output while the new value is written into the chosen memory location. However, if a single-port memory is employed, the read access has to be carried out one cycle prior to the write access to the same address, and hence an additional RAM word is needed to temporarily store the old value. Since the two-port constraint was also assumed in the competing hybrid architectures, the effect of an additional storage word is cancelled out.

V. Conclusion

We presented a new class of hybrid survivor path architecture based on RE and TF concepts. Latency and storage requirement can be traded for implementation complexity. To be specific, the RE complexity is lowered by employing denser storage cells. No partitioning is necessary for this memory, independent of the number of decoding blocks, contrary to combined TB and TF architectures. Therefore, our approach can be seen as means to extend the desirable high-speed low-latency feature of pure RE implementations even for a larger number of states. Furthermore, contrary to some other existing hybrid architectures, this new architecture is not bound to a specialized memory implementation and can thus be optimized for different platforms.

Acknowledgment

The authors thank the anonymous reviewers for their comments, which were invaluable in improving this brief.

References

[1] G. Feygin and P. G. Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," *IEEE Trans. Commun.*, vol. 41, no. 3, pp. 425–429, Mar. 1993.

[2] P. J. Black and T. H. Meng, "Hybrid survivor path architectures for Viterbi decoders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, Minneapolis, MN, Apr. 1993, pp. 433–436.

[3] E. Paaske, S. Pedersen, and J. Sparsø, "An area-efficient path memory structure for VLSI implementation of high speed Viterbi decoders," *Integr. VLSI J.*, vol. 12, no. 1, pp. 79–91, Nov. 1991.

[4] P. J. Black and T. H. Meng, "A 140-Mb/s, 32-state, radix-4 Viterbi decoder," *IEEE J. Solid-State Circuits*, vol. 27, no. 12, pp. 1877–1885, Dec. 1992.

[5] E. Boutillon and N. Demassieux, "A generalized precompiling scheme for surviving path memory management in Viterbi decoders," in *Proc. IEEE Int. Symp. Circuits Syst.*, Chicago, IL, May 1993, vol. 3, pp. 1579–1582.

[6] G. Fettweis, "Algebraic survivor memory management for Viterbi detectors," *IEEE Trans. Commun.*, vol. 43, pp. 2458–2463, Sep. 1995.

[7] J.-S. Han, T.-J. Kim, and C. Lee, "High performance Viterbi decoder using modified register-exchange methods," in *Proc. IEEE Int. Symp. Circuits Syst.*, Vancouver, Canada, May 2004, pp. 553–556.

[8] Y. Gang, A. T. Erdogan, and T. Arslan, "An efficient pre-traceback architecture for the Viterbi decoder targeting wireless communication applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 9, pp. 1918–1927, Sep. 2006.