ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/4089830

Pipelined parallel architecture for high throughput MAP detectors

Conference Paper · June 2004

DOI: 10.1109/ISCAS.2004.1329319 · Source: IEEE Xplore

CITATIONS
3

READS
14

3 authors, including:

Ruwan Ratnayake
London Metropolitan University

7 PUBLICATIONS **21** CITATIONS

SEE PROFILE

Gu-Yeon Wei
Harvard University

234 PUBLICATIONS **7,701** CITATIONS

SEE PROFILE

All content following this page was uploaded by Gu-Yeon Wei on 18 March 2015.

The user has requested enhancement of the downloaded file.

PIPELINED PARALLEL ARCHITECTURE FOR HIGH THROUGHPUT MAP DETECTORS

Ruwan Ratnayake, Gu-Yeon Wei and Aleksandar Kavčić

Division of Engineering and Applied Sciences
Harvard University, MA 02138, USA

ABSTRACT

A *maximum a posteriori probability* (MAP) detector based on a forward only algorithm with high throughput is considered in this paper. MAP gives the optimal performance and, with Turbo decoding, can achieve performance close to the channel capacity limits. Deep pipelined architecture for the forward only method is presented and compared with the other throughput-increasing methods. Simulation results based on the iterative MAP-LDPC (low-density parity check) system are shown. Hardware implementation issues that exploit the regularities of the structure are also discussed.

1. INTRODUCTION

High-speed detectors are of interest in research as well as in industry, particularly in magnetic recording where speeds on the order of 1Gps are needed. Naturally, proposed methods that perform in the Gbps range use computationally less intensive algorithms such as the Viterbi detector, which generate hard outputs [1]. Even though these detectors reach the 1Gbps milestone in throughput, their inherent inability to generate soft outputs make them less attractive for use in iterative systems. Thus, algorithms that give soft outputs such as the soft output Viterbi algorithm (SOVA) are attractive since they can exploit iterative detection for better performance [2]. However, these are still suboptimal algorithms in terms of bit error rate (BER) performance. Up to now, *maximum a posteriori* (MAP) algorithms that give optimal performance have not been considered for high-speed detectors due to their computational complexity. MAP detectors have so far only targeted wireless communication systems where data throughput requirements are much lower.

The MAP algorithm by Bahl, Cocke, Jelinek and Raviv (BCJR), requires forward and backward computations (FB-BCJR) [3]. This is in contrast to the Viterbi or SOVA algorithms, which allow the computations to be performed only in the forward direction. Once the input stream is fed into Viterbi/SOVA detectors, the output is generated after a fixed delay and retain the same order [4]. However, the *a posteriori probability* (APP) output of the FB-BCJR algorithm can only be evaluated after both forward and backward metrics are computed. Inevitably, the outgoing symbols appear in a permuted order relative to the incoming symbols.

There is a scheme that performs MAP with computations only in the forward direction [5]. We call this algorithm forward-only BCJR (FOBCJR). The data flow of this algorithm is similar to the Viterbi algorithm, where soft outputs are computed after a fixed delay relative to the incoming symbols, resulting in ordered outputs. Similar to the Viterbi algorithm, FOBCJR keeps track of *soft* survivors, which are kept in a fixed-length sliding-window survivor memory. A prominent feature of FOBCJR is its parallel structure, where as FB-BCJR only allow sequential state metric computations. Parallelism facilitates pipelining, resulting in an increase in throughput. In terms of throughput and input to output delay, FOBCJR with a deep pipelined structure is superior to other methods for computing APPs.

This paper begins with an overview of the FOBCJR algorithm. Afterwards in Section 3, we introduce three possible schemes to improve the throughput of the FOBCJR algorithm. By making quantitative comparisons we show that one of these methods, which is based on deep pipelined computations, outperforms the other two in terms of throughput and require less hardware. Section 4 presents the simulation results for these methods. Section 5 discusses some implementation methods that can exploit the inherent properties of the FOBCJR algorithm. Finally, Section 6 concludes the paper.

2. FORWARD ONLY BCJR

As implied by the name, FOBCJR computes state metrics only in the forward direction. It performs 3 basic tasks, namely *extend*, *update* and *collect*. This is similar to the Viterbi algorithm which performs extend, update and select. *Extend* and *update* are recursive operations. The *extend* operation extends the state buffer by adding one new column of state metrics based on the current received sample and previous state metrics. The collect operation extracts the APPs at the other end of the state buffer. The remaining state metrics of the buffer are updated with the *update* operation based on information on the same received sample. A key feature of FOBCJR is that the all *update* operations

are performed in parallel. This is in contrast to the sequential behavior of FB-BCJR. This parallelism enables *update* to be pipelined. For brevity, we defer the reader to [5] for a detailed discussion of the FOBCJR algorithm.

Extend, *update* and *collect* operations are essentially evaluating probabilities by combining probabilities of relevant paths. Simply, these are sum-product operations. Since computing products is expensive in terms of required computational power and complexity, the algorithm is implemented in log domain. Thus, a product is mapped to an addition and an addition is mapped to a special operation which is denoted by $\boxplus$. This special addition $\boxplus$ is defined as $a \boxplus b = \ln(e^a + e^b) = \max(a, b) + \ln(1 + e^{-|a-b|})$. The correction term, $\ln(1 + e^{-|a-b|})$, can be implemented as a one dimensional look up table (LUT). The bottleneck of a MAP algorithm is the state metric computation, consisting of adders and $\boxplus$ units, see Fig. 1. We call the unit in Fig. 1, the add/compare/select/LUT/add (ACSLA) unit.

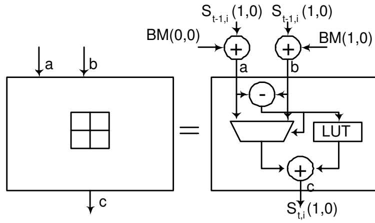

Fig. 1. ACSLA state metric computation unit for log MAP.

3. INCREASING THROUGHPUT

In this section, we investigate three different schemes to increase the throughput of the system. First, we can speed up the ACSLA unit by re-ordering operations. Second, we can unroll the trellis, which results in an increase in the delay of the critical path, but also increases throughput. Finally, we propose a simplified *extend* operation and deep pipelined *update* operation version of the algorithm and compare its merits to the other two methods.

3.1. Compare Select Add LUT Add (CSALA)

Lee *et al.* show that the critical path in the Viterbi algorithm can be increased by reorganizing the add/compare /select (ACS) unit to a compare/select/add (CSA) unit [6]. Similarly, the ACSLA unit for an APP algorithm can be reorganized such that addition and comparison are performed in parallel and is illustrated in Fig.2. This is referred to as a compare/select /add/LUT/add (CSALA) unit. If both *extend* and *update* operations are based on CSALA, then the delay in the critical path of the system is reduced by the delay for one addition operation.

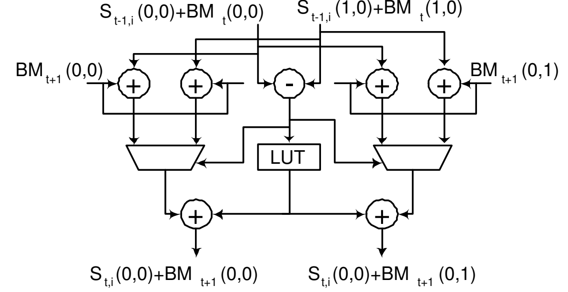

Fig. 2. CSALA *extend*, *update* computation unit

3.2. Radix 4-Sum2

Another approach is to unroll the trellis to increase throughput. We consider a radix-4 system where two observation samples are considered within one cycle. The aim is that time taken for the state updates with this method would be less than twice the time taken for state updates in a radix-2 system. This method has been exploited to increase throughput in Viterbi systems [7]. However, this scheme is not very attractive for MAP since four paths must be combined together instead of merely obtaining the maximum of four paths as in Viterbi. Combining four paths requires a three-ACSLA-unit tree with two levels and the delay through the critical path is only one addition less than twice the delay for ACSLA.

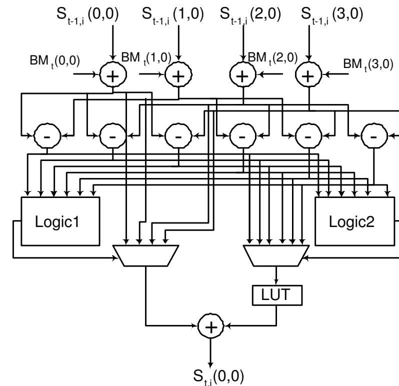

Fig. 3. Radix4 Sum2 *extend*, *update* computation unit

The combining of 4 paths in radix-4 is given by $a \boxplus b \boxplus c \boxplus d = \max(a, b, c, d) + \Delta$ where the correction term, Δ, is not as simple an expression for implementation. However, if the two most significant paths are combined instead of all four paths, then the state update becomes simpler. Assume mx_1 and mx_2 are the first two maximums among the four paths, then metric update becomes, $mx_1 \boxplus mx_2 = \max(mx_1, mx_2) + \ln(1 + e^{-|mx_1 - mx_2|})$. This computation is similar to radix-2 state updates. This new scheme is shown in Fig.3 (for a 4-state trellis system) and called radix4-sum2 since it is based on radix-4 and effectively com-

bines two paths. There are six possible comparisons for four values. *Logic1* and *Logic2* compute the maximum value and the difference of the first two maximum values based on the six comparisons, respectively.

The delay of this new scheme would be only moderately larger than the radix 2 system (due to the logic operation) and since two symbols are decoded within one cycle the throughput is increased compared to ACSLA. The effect of combining two paths instead of four paths on BER performance is discussed in Section 4 and is shown to be marginal.

3.3. CSA-Extend and Deep Pipelined Update

There is yet another method to increase the throughput. Fig.4 illustrates the data flow for *extend* and *update* operations in the FOBCJR algorithm. *Extend* is a feed back loop. In other words, the value of the state register for a particular time instance depends on the value of the same register in the previous time instance. This prevents one from pipeling the *extend* operation. On the other hand, all states pertaining to the *update* operation are performed in parallel and can be pipelined. Therefore, the *extend* operation is the critical path. Reducing this delay and using shorter pipe-stage delays in the *update* part can increase the overall throughput.

In order to reduce the *extend* delay, we can make a simplification by ignoring the correction term, $\ln(1 + e^{-|a-b|})$, when combining two paths. This effectively reduces the *extend* operation to a simple ACS operation. We can further reduce the delay of *extend* by reordering the ACS operation to a CSA operation. The *update* part is a deep pipelined ACSLA operation. Thus, we can increase throughput by reducing the *extend* delay and reducing the delay of each pipe stage in the *update* part at the expense of adding more pipe stages. The effect of using a simplified *extend* operation on BER is discussed in Section 4 and shown to be marginal.

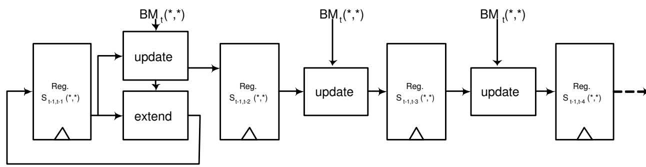

Fig. 4. Block diagram of data flow of FOBCJR

3.4. Comparison of the Schemes

Table 1 presents a comparison of these three methods in terms of the approximate number of computational units and storage devices. The values are derived based on a 16 state E2PR4 channel with a sliding window length of 24. All units are 7-bit wide computing elements. Delay of the sequential operation through the critical path is also shown for each method. The normalized throughput (CSALA normalized to 1) is based on the assumption that delay through an adder is twice the delay through a multiplexer (MUX) or a LUT.

Table 1. Comparison of proposed schemes

Method	Add	MUX	LUT	Reg. 7bit	Delay - Critical Path	ThruPut (norm)
CSALA*	4600	1570	1570	1540	$2D_A + D_{LUT}$	1
Rdx4-Sum2*	8600	1550	1500	770	$3D_A + D_{Logic} + D_{LUT} + D_S$	1.11
CSA-Deep Pipe*	3200	850	820	1000	$D_A + D_{MUX}$	1.66
FB-BCJR	520	130	130	7500	$3D_A + D_{LUT}$	1.07

D_A : Delay of a adder, D_{LUT} : Delay of LUT, D_S : Delay of a MUX
* FOBCJR

For comparison purposes, a high-throughput version of the FB-BCJR is included. This FB-BCJR has two forward and two backward metric calculation units that run in parallel. The effective throughput with this setting is $3/2$ times the rate of these state updates.

Even though the CSALA and radix-4-sum2 methods improve throughput compared to a ACSLA-based FOBCJR, they require significantly more computational units compared to other methods. Among the three FOBCJR methods, CSA-*extend*-deep pipelined-*update* requires less hardware and enables higher throughput.

It is evident that FOBCJR base on CSA-*extend* deep pipelined-*update* is computationally intensive compared to FB-BCJR. On the other hand, FB-BCJR is more storage intensive and has lower throughput. Moreover, FB-BCJR needs complex control mechanisms to control the forward, backward computational units and to combine their results appropriately.

4. SIMULATION RESULTS

BER performance for the proposed methods, namely radix-4-sum2 and CSA-*extend* deep pipelined-*update* method for FOBCJR are shown in Fig. 5. The system consists of combining a FOBCJR detector combined with a low-density parity check (LDPC) decoder for iterative decoding. The detector targets an E2PR4 channel model with 16 states. The FOBCJR takes in 7-bit soft inputs and outputs 7-bit extrinsic information. Frame length of 5120 bits with code rate of 0.8 and window length of 24 are considered. The BER results are for 20 LDPC iterations, where for every 5 LDPC iterations detector performs one iteration.

With these parameters, maxlogMAP, which is a suboptimal FB-BCJR algorithm, has about 0.15dB degradation in performance relative to logMAP at BER 10^{-4}. The two proposed methods to increase throughput for FOBCJR, radix4-sum2 and CSA-*extend*-deep pipelined-*update*, have negligible difference in BER performance compared to logMAP.

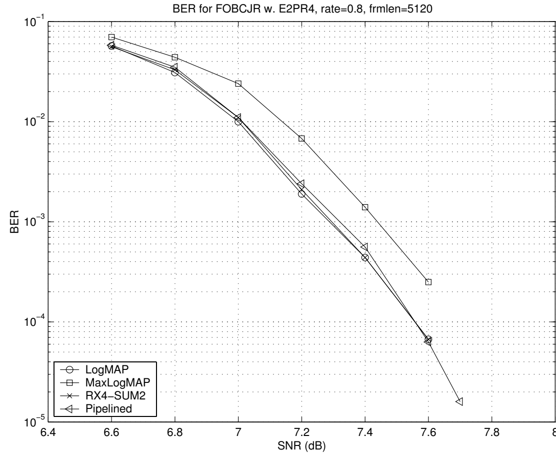

Fig. 5. BER results for logMAP, MaxlogMAP, Radix4-Sum2 and CSA-*extend* deep pipelined-*update* methods.

5. IMPLEMENTATION ISSUES

Fig 6 illustrates the proposed CSA-*extend* deep pipelined-*update* method for FOBCJR for a system with 2 states. It is evident that the entire *updating* part has a very regular structure. This regularity in the structure makes it possible to design a single basic block with embedded connections and replicate it as required by the window length.

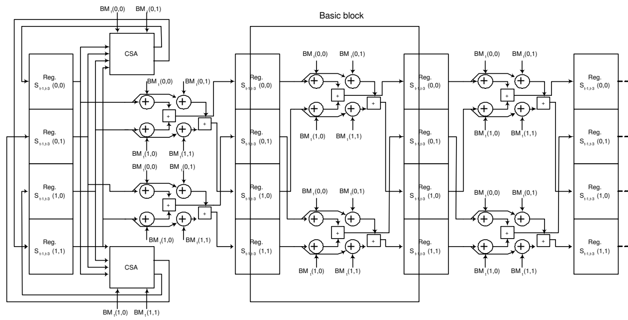

Fig. 6. CSA-*extend* deep pipelined-*update* FOBCJR

One of the issues with large numbers of states in the trellis is that the vertical height of the layout can be considerably large. Moreover, in a typical trellis structure, large numbers of wires crisscross when connecting the required states. In general, for a trellis with M states there are $M-1$ crisscrosses. Wiring in this manner would require significant area. Fig. 7(a) shows trellis connections for a 4 state system. It is possible to reduce the vertical height and the number of crisscrosses with efforts to connect more states with horizontal wires by modifying the ordering of the states. Fig. 7(b) illustrates this ordering for the same system. State registers laid out according to this manner would greatly reduce wiring area of the chip when implementing FOBCJR.

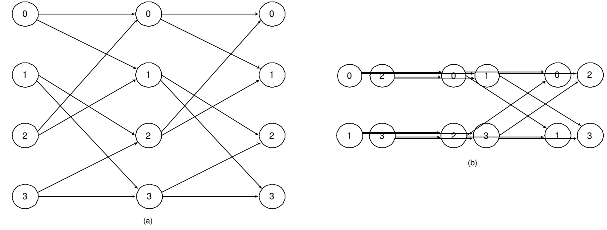

Fig. 7. Reordering of states in the trellis.

6. CONCLUSION

This paper compared three methods to increase the throughput of forward only BCJR(FOBCJR), which is an optimal MAP algorithm. We propose a FOBCJR algorithm with simplified *extend* part and deep pipelined *update* part that can increase throughput with less hardware requirements compared to other methods. Simulation results show that simplification on the *extend* part has negligible effect on BER performance. We showed how embedded wiring and replication can be used to implement this algorithm. The FOBCJR algorithm with simplified *extend* and deep pipelined *update* is currently under design and will be fabricated in a 0.13μm CMOS technology.

7. REFERENCES

[1] P. J. Black and T. H. Y. Meng, “A 1-Gb/s, Four-State, Sliding Viterbi Decoder,” *IEEE J. Solid-State Circuits,* vol. 32, pp. 797-805, Jun. 1997.

[2] E. Yeo, S. A. Augsberger, W R. Davis and B. Nikolic, “ A 500-Mb/s Soft-Outpot Viterbi Dcoder,” *IEEE J. Solid-State Circuits,* vol. 38, pp. 1234-1241, Jul. 2003.

[3] L.R. Bahl, J. Cocke, F. Jelinek and J. Raviv, “Optimum Decoding of Linerar Codes for Minimizing Symbol Error Rate,” *IEEE Trans. Inform. Theory,* vol. 13, pp. 284-287, Mar. 1974.

[4] J. Hagenauer, “Source-Controlled Channel Decoding,” *IEEE Trans. Commun.,* vol. 43, pp. 2449-2457, Sep. 1995.

[5] X.Ma and A. Kavcic, “Path Partition and Forward-Only Trellis Algorithm,” *IEEE Trans. Inform. Theory,* vol. 49, pp. 38-52, Jan. 2003.

[6] I. Lee and J. L. Sonnatag, “A New Architecture For The Fast Viterbi Algorithm,” *Proc. IEEE GLOBECOM,* pp. 1664-1668, San Francisco, CA, Nov 2000.

[7] P. J. Black and T. H. Y. Meng, “A 140-Mb/s, 32 State, Radix-4 Viterbi Decoder,” *IEEE J. Solid-State Circuits,* vol. 27, pp. 1877-1885, Dec. 1992.

View publication stats