DESIGN METHODOLOGY FOR HIGH-SPEED ITERATIVE DECODER ARCHITECTURES

Mohammad M. Mansour and Naresh R. Shanbhag

Coordinated Science Laboratory/ECE Department
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801
[mmansour,shanbhag]@mail.icims.csl.uiuc.edu

ABSTRACT

We propose a novel approach to the design and analysis of VLSI architectures for the soft-input soft-output *a posteriori* probability (SISO-APP) decoding algorithm used in iterative decoders such as turbo decoders. The approach is based on a *tile-graph* composed of recursion patterns that model the resource-time scheduling of the forward-backward recursion equations of the algorithm. The problem of constructing a SISO-APP architecture is formulated as a three-step process of constructing and counting the patterns needed and then tiling them. The problem of optimizing the architecture for high speed and low power reduces to optimizing the individual patterns and the tiling scheme for minimal delay and storage overhead. The various forms of the sliding and parallel-window (PW) architectures in the literature are instances of the proposed tile-graph. Using the tile-graph approach, a new PW architecture controlled by the window width r is proposed that achieves for $r = 10$ a 45%, a 71%, a 51%, and a 25% reduction in decoding delay, state, input, and output metrics storage respectively, compared to a conventional architecture with a 10% increase in resources.

1. INTRODUCTION

It is known that turbo codes [1] and related concatenated convolutional codes [2] are capable of achieving near Shannon-error correction capability at least on binary symmetric and AWGN channels. This breakthrough in performance is attributed to the concept of iterative soft information exchange among constituent decoders in a decoding network. The core of these constituent decoders is the BCJR *a posteriori* probability (BCJR-APP) decoding algorithm [3] or the related soft-input soft-output APP (SISO-APP) algorithm [4]. These APP decoding algorithms suffer a high inherent latency and substantial storage requirements due to the serial recursion bottleneck involved in the processing of key equations of the algorithms which limits their applicability in latency and power sensitive applications such as wireless communications and portable computing. The added requirements of (de)interleaving and multiple decoding stages involved in iterative decoding impose further challenges for a practical VLSI implementation to meet the required energy/delay/storage constraints.

Several approximations to the APP algorithm have been proposed that attempt to break the serial processing bottleneck such as the sliding-window (SW) approximation [4], the warmup-valid metric recursion approximation [5], and the single/double flow recursion approximations [6]. Other approximations attempted to mitigate both effects of high latency and storage requirements by applying the recursion approximations to both the forward and backward recursion equations which allowed independent processing of portions of the input frame in parallel [7]. Approximations related to reducing the computational complexity of the algorithm were proposed in [8]. Practical and efficient SISO-APP architectures rely heavily on the effect of the these approximations both on delay, storage, power consumption on one hand and communications performance on another hand. However, there does not exist a simple and systematic methodology to perform tradeoffs between VLSI performance and communications performance using these approximations or possibly to propose other approximations that can improve both performance aspects. Such a methodology is valuable for a VLSI designer in the early design stages.

In this paper we propose a graphical design and analysis approach based on a tile-graph that models the effects of latency and storage of the forward-backward recursion equations of the SISO-APP algorithm on a resource-time graph [9]. The tile-graph is composed of multiple recursion patterns tiled together, where each recursion pattern is assigned a portion of the decoding task. Tradeoffs related to latency, storage requirements, and communications performance of the whole architecture as represented by the tile-graph are then based of the performance of these recursion patterns and the way they are tiled together. The attractive features of this approach are twofold. First, as an analysis method, it is simple, systematic and yet general enough

This work was supported with funds from NSF under grants CCR 99-79381 and CCR 00-73490.

to evaluate the architectural effects of the above mentioned approximations. Second, as a design method, it is relatively easy to construct a tile-graph having a certain delay and storage constraint from tiled recursion patterns by optimizing the patterns and the tiling scheme to meet the constraints. The SW, single/double flow, and parallel-window (PW) architectures proposed in [4]-[6] are instances of the tile-graph having a specific tiling scheme, and can be systematically analyzed by the tile-graph approach.

Section II of the paper briefly introduces the SISO-APP algorithm [4]. Section III proposes the tile-graph as an design and analysis approach of SISO-APP decoder architectures. In section IV, an optimized PW architecture based on the tile-graph approach is proposed. Finally, Section V concludes the paper.

2. THE SISO-APP DECODING ALGORITHM

The SISO-APP algorithm [4] is a probabilistic decoding algorithm of codes that can be described by a trellis. The SISO-APP algorithm takes as input likelihood ratios (LLR's) associated with the data and code symbols and generates extrinsic posterior LLR's according to the inputs and the code constraints. The output of the algorithm is the "soft information" to be exchanged in an iterative decoding procedure. Let ($\lambda_k[\boldsymbol{uc}] = \lambda_k[\boldsymbol{u}] + \lambda_k[\boldsymbol{c}]$) denote the combined input LLR's (also called branch metrics) and ($\Lambda_k[\boldsymbol{c}]$, $\Lambda_k[\boldsymbol{u}]$) denote the output LLR's of all data symbols $\boldsymbol{u}$ and code symbols $\boldsymbol{c}$ of the code. The computations of $\Lambda_k[\cdot]$ are performed in two phases. In phase I, two auxiliary quantities ($\alpha_k[s], \beta_k[s]$) called the forward and backward state metrics, respectively, are computed recursively for all states in the trellis by traversing the trellis in two opposite directions. In each direction, metrics are accumulated from state to state using all connecting trellis edges as:

$$\alpha_k[s_j] = \max_{e:E(e)=s_j}{}^{*} \{\alpha_{k-1}[S(e)] + \lambda_k[\boldsymbol{uc}(e)]\}, \quad (1)$$

$$\beta_{k-1}[s_i] = \max_{e:S(e)=s_i}{}^{*} \{\beta_k[E(e)] + \lambda_k[\boldsymbol{uc}(e)]\}, \quad (2)$$

where $k = 1, \cdots, L-1$ in (1) and $k = L, \cdots, 2$ in (2), L is the frame length, s_j and s_i are trellis states, e is a trellis edge connecting two states with $S(e)$ and $E(e)$ denoting respectively the starting and ending states of e, $\boldsymbol{uc}(e)$ is the concatenation of the data and code symbols labelling e. The function max* is defined as $\max^*\{x, y\} = \max\{x, y\} +$ a correction factor [8]. The recursions are initialized with $\alpha_0[s_0] = \beta_{L-1}[s_0] = 0$, and $-\infty$ otherwise.

In phase II, the output LLR's are generated using the auxiliary functions of phase I as:

$$\Lambda_k[\boldsymbol{x}] = \max_{e:\boldsymbol{x}(e)=\boldsymbol{x}}{}^{*} \{\alpha_{k-1}[S(e)] + \lambda_k[\boldsymbol{uc}(e)] + \beta_k[E(e)]\}, \quad (3)$$

where $\boldsymbol{x} = \boldsymbol{u}$ or $\boldsymbol{c}$. Equations (1)-(3) are called the key equations of the SISO-APP algorithm.

The computations and flow of data involved in the key equations can be modelled graphically on a dataflow graph (DFG). DFG's provide flexibility in exploiting resource-time tradeoffs without impacting the design style. Moreover, optimizations can be easily exposed on a DFG through dataflow analysis [6]. The resources consist of the following types: an α-metric processing unit (α-MPU) that performs α-metric computations of (1), a β-MPU that performs the β-metric computations of (2), and a Λ-MPU that generates the output reliabilities of (3). Figure 1 shows the DFG of the SISO-APP algorithm with the time index running from top to bottom. The α-metrics are computed and stored from left-to-right by the α-MPU's from time 0 to $L-1$ (shaded region in Fig. 1). At time $L-1$, output reliabilities are produced from right to left, in reverse order with respect to the trellis sections, by the β, Λ-MPU's using the stored α-metrics and the initial β-metrics, then the β-metrics are updated. Decoding delay is proportional to the height of the graph, and storage requirements are proportional to the are of the shaded region in the graph. The primary objective is to perform dataflow optimizations on the DFG in Fig. 1 to minimize both delay and storage requirements.

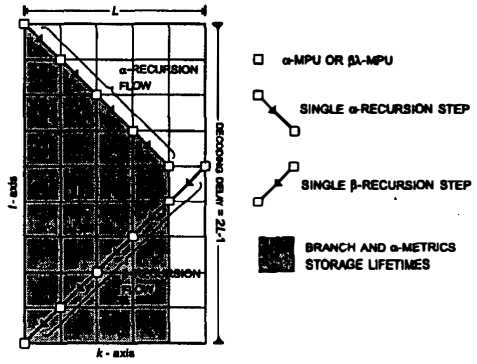
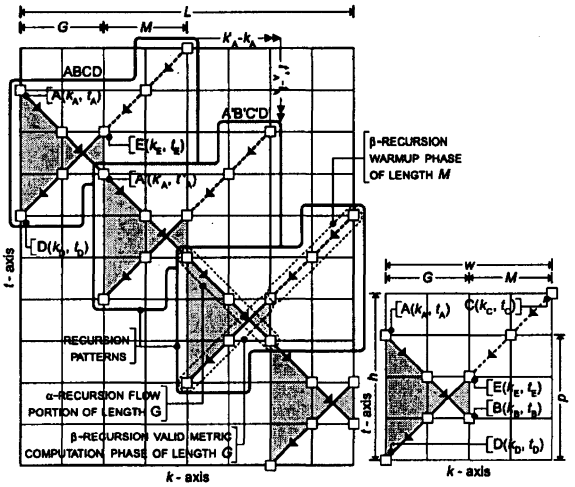


Fig. 1. The DFG of the SISO-APP algorithm.

3. THE TILE-GRAPH APPROACH

We propose to divide the DFG that decodes L code symbols into smaller flow graphs, called *recursion patterns*, where each recursion pattern decodes $G < L$ code symbols, and optimize these recursion patterns to come up with an architecture that incurs minimal decoding delay and requires the least metric storage area. An optimized DFG can then be constructed using the optimized recursion patterns. The α and/or β-recursion flows can be broken down into smaller portions, where each portion is composed of a metric warmup phase of length M followed by a valid metric computation phase of length G. Appropriate portions from both

recursion flows can be paired to form a recursion pattern that decodes G symbols. As an example, Fig. 2(a) shows how the sliding-window [4] DFG (SW-DFG) can be decomposed into recursion patterns identical to pattern ABCD shown in Fig. 2(b). Here, only the β-recursion flow is broken into smaller portions. Conversely, the SW-DFG can be constructed by *tiling* the recursion patterns diagonally. A DFG constructed by tiling recursion patterns is called a *tile-graph*. Consequently, minimizing delay and storage area of a DFG translates to finding the most compact diagonal tiling of the individual recursion patterns.



(a) Tile-graph (b) Recursion pattern ABCD

Fig. 2. The tile-graph representation of the SW-DFG: (a) Decomposition into recursion patterns. (b) A recursion pattern ABCD with parameters (d, M, G) defined by the points A, B, C, D.

3.1. Tile-Graph Construction

We consider the process of constructing the tile-graph of the SISO-APP algorithm when the sliding-window approximation is applied. A recursion pattern of a sliding window architecture of the SISO-APP algorithm can be configured using three parameters (d, M, G) [9], where (refer to Fig. 2): 1) $d = t_A - t_B$ is the difference between the starting time (t_A) of the α-recursion flow and the ending time (t_B) of the β-recursion flow, 2) M is the number of metric computations needed to initialize the β-recursion flow (also called the metric warmup depth) which affects communications performance, and 3) G is the number of *valid* metric computations performed by the β-recursion flow. Then the problem of constructing the SW-DFG can be summarized in three steps. In **step 1**, a single recursion pattern with parameters (d, M, G) is constructed. It can be shown that the coordinates of A, B, C, D of a pattern ABCD (e.g. see Fig 2(b)) can be uniquely determined from (d, M, G), and that only six feasible patterns exist [9]. In **step 2**, the number of patterns is determined from (d, M, G) and L as $\lceil \frac{L}{G} \rceil$. Finally, in **step 3**, the patterns are tiled diagonally. The tiling separation between adjacent patterns needs to be determined (see horizontal and vertical lines with double arrowheads in Fig. 2(a)). It can be shown that the horizontal and vertical offsets are given by G and $\frac{|d|+|d-2G|}{2}$, respectively.

3.2. Tile-Graph Analysis

Using the tile-graph representation of an architecture for the SISO-APP algorithm, it is easy to analyze the performance of the architecture in terms of decoding delay and storage requirements. The decoding delay of a recursion pattern is the delay incurred by metric computations as a result of the flow of operations represented by the pattern, or geometrically, the height of the pattern. The total delay of the architecture can be determined from the delay of the recursion patterns and the tiling scheme. For the SW-DFG, the delay between two adjacent patterns is the vertical offset between them, resulting in a total decoding delay of [9]

$$\tau_d = \left(\frac{L}{G} - 1\right)\left(\frac{|d| + |d - 2G|}{2}\right) + \frac{2G + M + |d - G| + |d - G - M|}{2}. \quad (4)$$

The total metric storage lifetimes $\tau_{m_\alpha + m_\beta}$, for both α and β, of a pattern ABCD, is the area of the region defined by the points A, B, E, D (see dark grey regions in Fig. 2) [9]

$$\tau_{m_\alpha + m_\beta} \triangleq \tau_{m_\alpha} + \tau_{m_\beta} = \frac{L}{G} \cdot \sum_{i=0}^{G} |d - 2G + 2i|. \quad (5)$$

Equations (4) and (5) characterize the performance of any sliding window architecture of the SISO-APP algorithm in terms of the parameters (d, G, M) and L. It can be shown [9] that the optimum value of d that jointly minimizes both τ_d and $\tau_{m_\alpha + m_\beta}$ is $d^* = G$. Similarly, a recursion pattern with $d = G$ has minimum delay and storage requirements.

4. OPTIMIZED SISO-APP ARCHITECTURES

We propose a new optimized parallel-window DFG (PW-DFG) for the SISO-APP algorithm based on the tile-graph approach. First a tile-graph using a parallel tiling scheme is constructed (Fig. 3), where both the α and β-recursion flows are broken into smaller portions and the recursion pattern constructed has $d = G$. By analyzing the tile-graph in Fig. 3, it is obvious that there is a significant improvement in decoding delay over the SW-DFG in Fig. 2(a). However, the metrics storage area is still high (see area of shaded regions in Fig. 3). It can be shown that the recursion pattern

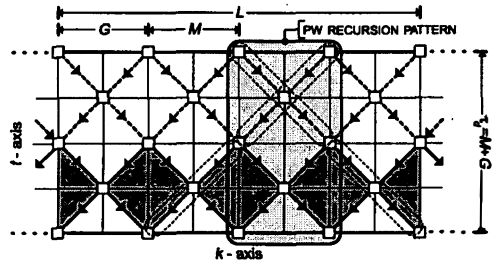


Fig. 3. The tile-graph representation of the PW-DFG.

in Fig. 3 is not efficient in terms of metric reuse across adjacent windows, as well as in terms of state metrics storage area. Assuming that the width of the recursion pattern is $G = rM$, the proposed PW recursion pattern is based on splitting the recursions of the pattern further until a desired level of decoding delay and metrics storage overhead is reached. For example, Fig. 4 shows the result of splitting the recursions of the pattern in Fig. 3 two times. Also shown in the figure are the storage needed to align the input and output metrics. The decoding delay, total state metrics storage, input and output metrics storage are proportional to $M(\frac{r}{2}+1)$, $\frac{rM}{8}+\frac{4}{r}+2$, $\frac{5rM}{16}+M$, and $\frac{3rM}{16}$, respectively. For a width of $r = 10$, the proposed architecture achieves a 45%, a 71%, a 51%, and a 25% reduction in decoding delay, and state, input, and output metrics storage respectively, over the architecture in Fig. 3, at the expense of a 10% increase in the number of MPU's.

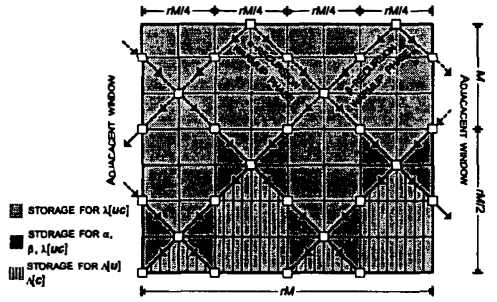


Fig. 4. A PW pattern with recursion splitting applied twice.

5. CONCLUSION

In this paper, a novel tile-graph approach to the analysis and design of architectures for the SISO-APP algorithm was proposed. As a design method, the tile-graph approach can be used to construct architectures using tiled recursion patterns where each pattern is assigned a portion of the decoding task. As an analysis method, it is a simple and systematic approach to evaluate architectural tradeoffs related to latency and storage requirements. The sliding and parallel-window architectures in the literature are shown to be instances of a tile-graph. Further, a new PW-DFG based on the tile-graph approach was proposed that is capable of achieving a desired decoding delay and storage overhead by splitting the recursions and tuning the window width.

6. REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *IEEE Int. Conf. on Communications*, 1993, pp. 1064–1070.

[2] S. Benedetto et al., "Soft-input soft-output modules for the construction and distributed iterative decoding of code networks," *European Trans. Telecomun.*, vol. ETT 9, pp. 155–172, March/April 1998.

[3] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Tran. on Information Theory*, pp. 284–287, Mar. 1974.

[4] S. Benedetto et al., "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes," TDA Progress Report 42-127, JPL, November 1996.

[5] H. Dawid and H. Meyr, "Real-time algorithms and VLSI architectures for soft output MAP convolutional decoding," in *Personal, Indoor and Mobile Radio Communications, PIMRC'95. Wireless: Merging onto the Information Superhighway*, 1995, vol. 1, pp. 193–197.

[6] C. Schurgers, F. Catthoor, and M. Engels, "Energy efficient data transfer and storage organization for a MAP turbo decoder module," in *Proc. International Symposium on Low Power Electronics and Design*, 1999, pp. 76–81.

[7] A. Worm, H. Lamm, and N. Wehn, "VLSI architectures for high-speed MAP decoders," in *Fourteenth International Conference on VLSI Design*, 2001, pp. 446–453.

[8] P. Roberston, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operatiting in the log domain," in *IEEE Int. Conf. on Communications*, 1995, pp. 1009–1013.

[9] M. Mansour and N. Shanbhag, "VLSI architectures for iterative decoders," *submitted to IEEE Transactions on VLSI Systems*, 2001.