A Low Latency SISO with Application to Broadband Turbo Decoding

Peter A. Beerel, *Member, IEEE,* and Keith M. Chugg, *Member, IEEE*

***Abstract*—The standard algorithm for computing the soft-inverse of a finite-state machine [i.e., the soft-in/soft-out (SISO) module] is the forward–backward algorithm. These forward and backward recursions can be computed in parallel, yielding an architecture with latency $\mathcal{O}(N)$, where N is the block size. We demonstrate that the standard SISO computation may be formulated using a combination of prefix and suffix operations. Based on well-known tree-structures for fast parallel prefix computations in the very large scale integration (VLSI) literature (e.g., tree adders), we propose a tree-structured SISO that has latency $\mathcal{O}(\log_2 N)$. The decrease in latency comes primarily at a cost of area with, in some cases, only a marginal increase in computation. We discuss how this structure could be used to design a very high throughput turbo decoder or, more generally, an iterative detector. Various subwindowing and tiling schemes are also considered to further improve latency.**

***Index Terms*—Iterative detection/decoding, parallel prefix computations, turbo coding.**

I. Introduction

CALCULATING the "soft-inverse" of a finite-state machine (FSM) is a key operation in many data detection/decoding algorithms. Perhaps the most appreciated application is iterative decoding of concatenated codes, such as turbo codes [1], [2]. However the soft-in/soft-out (SISO) module [3] is widely applicable in iterative and noniterative receivers and signal processing devices (e.g., [4]–[7]). The soft-outputs generated by a SISO may also be thresholded to obtain optimal hard decisions (e.g., producing the same decisions as the Viterbi algorithm [8] or the Bahl algorithm [9]). The general trend in many applications is toward higher data rates and, therefore, fast algorithms and architectures are desired.

There are two performance (speed) aspects of a data detection circuit architecture that are relevant to this paper. The first is *throughput* which is a measurement of the number of bits per second the architecture can decode. The second is *latency* which is the end-to-end delay for decoding a block of N bits. *Nonpipelined* architectures are those that decode only one block at a time and for which the throughput is simply N divided by the latency. *Pipelined* architectures, on the other hand, may decode multiple blocks simultaneously shifted in time [10], thereby achieving much higher throughput than their nonpipelined counterparts.

Depending on the application, the throughput and/or latency of the data detection hardware is important. For example, the latency associated with interleaving in a turbo-coded system with relatively low data rate (less than 100 kb/s) will likely dominate the latency of the iterative decoding hardware. For future high-rate systems, however, the latency due to the interleaver may become relatively small, making the latency of the decoder significant. While pipelined decoders [10] can often achieve the throughput requirements, such techniques generally do not substantially reduce latency. In addition, sometimes latency has a dramatic impact on overall system performance. For example, in a data storage system (e.g., magnetic hard drives), latency in the retrieval process has a dramatic impact on the performance of the microprocessor and the overall computer. Such magnetic storage channels use high-speed Viterbi processing with turbo-coded approaches suggested recently [11], [12].

The standard SISO algorithm is the forward–backward algorithm. The associated forward and backward recursion steps can be computed in parallel for all of the FSM states at a given time, yielding an architecture with $\mathcal{O}(N)$ computational complexity and latency, where N is the block size. The key result of this paper is the reformulation of the standard SISO computation using a combination of prefix and suffix operations, which leads to an architecture with $\mathcal{O}(\lg N)$ latency.[1] This architecture is based on a well-known tree-structure for fast parallel prefix computations in the very large scale integration (VLSI) literature (e.g., fast adders [13], [14]), so we refer to it as a *tree-SISO*.

This exponential decrease in latency for the tree-SISO comes at the expense of increased computational complexity and area. The exact value of these costs depends on the FSM structure (e.g., the number of states) and the details of the implementation. However, for a four-state convolutional code, such as those often used as constituent codes in turbo codes, the tree-SISO architecture achieves $\mathcal{O}(\lg N)$ latency with computational complexity of $\mathcal{O}(N \lg N)$. Note that, for this four-state example, the computational complexity of tree-SISO architecture increases sublinearly with respect to the associated speedup. This is better than well-studied linear-scale solutions to the Viterbi algorithm (e.g., [15]); the generalization of which to the SISO problem is not always clear. For this four-state code example, the area associated with the $\mathcal{O}(\lg N)$-latency tree-SISO is $\mathcal{O}(N)$.

After formally defining the SISO and prefix–suffix operations in Section II, we describe the reformulation and corresponding tree-SISO architecture in Sections III and IV, respectively. Compatibly of the tree-SISO with known latency reduction methods

Manuscript received May 3, 2000; revised December 6, 2000. This work was supported in part by the National Science Foundation under NCR-CCR-9726391. Ths work was presented in part at MILCOM 2000, Los Angeles, CA, October 2000.

The authors are with the Electrical Engineering – Systems Department, University of Southern California, Los Angeles, CA 90089-2565 USA (e-mail: pabeerel@usc.edu; chugg@usc.edu).

Publisher Item Identifier S 0733-8716(01)04187-7.

[1]We use $\lg$ to denote $\log_2$.

is discussed in Section V. We conclude with a discussion of the architecture's potential applications, feasibility, and performance given current VLSI trends. Detailed computational complexity and hardware analyses appear in Appendices I and II.

II. Background

A. SISO Modules

For concreteness, we consider a specific class of finite state machines with no parallel state transitions and a generic S-state trellis. Such a trellis has up to S transitions departing and entering each state. The FSM is defined by the labeling of the state transitions by the corresponding FSM input and FSM output. Let $t_k = (s_k, a_k, s_{k+1}) = (s_k, a_k) = (s_k, s_{k+1})$ be a trellis transition from state s_k at time k to state s_{k+1} in response to input a_k. Since there are no parallel state transitions, t_k is uniquely defined by any of these representations. Given that the transition t_k occurs, the FSM output is $x_k(t_k)$.[2]

Consider the FSM as a system that maps a digital input sequence a_k to a digital output sequence x_k. A marginal soft-inverse, or SISO, of this FSM can be defined as a mapping of soft-in (SI) information on the inputs $\mathrm{SI}(a_k)$ and outputs $\mathrm{SI}(x_k)$, to soft-output (SO) information for a_k and/or x_k. The mapping is defined by the combining and marginalization operators used. It is now well-understood that one need only consider one specific reasonable choice for marginalization and combining operators and the results easily translate to other operators of interest [13, Section 26.4], [16]–[18]. Thus, we focus on the min-sum marginalization-combining operation with the results translated to max-product, sum-product, min*-sum, and max*-sum [19] in the standard fashion. In all cases, let the indices K_1 and K_2 define the time boundaries of the *combining window* or span used in the computation of the soft-output for a particular quantity u_k [e.g., $u_k = s_k$, $u_k = a_k$, $u_k = t_k$, $u_k = x_k$, $u_k = (s_k, s_{k+d})$, etc.].[3] For min-sum marginalization-combining, the *minimum sequence metric (MSM)* of a quantity u_k is the metric (or length) of the shortest path or sequence in a combining window or span that is consistent with the conditional value of u_k. Specifically, the MSM is defined as[4]

$$\mathrm{MSM}_{K_1}^{K_2}(u_k) \triangleq \min_{\mathbf{t}_{K_1}^{K_2}:\, u_k} M_{K_1}^{K_2}\left(\mathbf{t}_{K_1}^{K_2}\right), \tag{1}$$

$$M_{K_1}^{K_2}\left(\mathbf{t}_{K_1}^{K_2}\right) \triangleq \sum_{m=K_1}^{K_2} M_m(t_m) \tag{2}$$

$$M_m(t_m) \triangleq \mathrm{SI}(a_m) + \mathrm{SI}(x_m(t_m)) \tag{3}$$

where the set of transitions starting at time K_1 and ending at time K_2 that are consistent with u_k is denoted $\mathbf{t}_{K_1}^{K_2}$: u_k and $\mathbf{t}_{K_1}^{K_2}$ implicitly defines a sequence of transitions $t_{K_1}, t_{K_1+1}, \ldots, t_{K_2}$. Depending on the specific application, one or both of the following "extrinsic" quantities will be computed

$$\mathrm{SO}_{K_2}^{K_1}(x_k) \triangleq \mathrm{MSM}_{K_1}^{K_2}(x_k) - \mathrm{SI}(x_k) \tag{4}$$

$$\mathrm{SO}_{K_1}^{K_2}(a_k) \triangleq \mathrm{MSM}_{K_1}^{K_2}(a_k) - \mathrm{SI}(a_k). \tag{5}$$

Because the system on which the SISO is defined is an FSM, the combining and marginalization operations in (1) and (2) can be computed efficiently. The traditional approach is the forward–backward algorithm which computes the MSM of the states recursively forward and backward in time. Specifically, for the standard *fixed-interval* algorithm based on soft-in for transitions t_k, $k = 0, 1, \ldots, N-1$, we have the following recursion based on add-compare-select (ACS) operations

$$f_k(s_{k+1}) \triangleq \mathrm{MSM}_0^k(s_{k+1}) \tag{6}$$

$$= \min_{t_k:\, s_{k+1}} [f_{k-1}(s_k) + M_k(t_k)] \tag{7}$$

$$b_k(s_k) \triangleq \mathrm{MSM}_k^{N-1}(s_k) \tag{8}$$

$$= \min_{t_k:\, s_k} [b_{k+1}(s_{k+1}) + M_k(t_k)] \tag{9}$$

where $f_{-1}(s_0)$ is initialized according to available edge information and $b_N(s_N)$ is set equal to a constant for each state s_N.[5] Note that, since there are S possible values for the state, these state metrics can be viewed as $(S \times 1)$ vectors $\mathbf{f}_k$ and $\mathbf{b}_k$. The final soft-outputs in (4) and (5) are obtained by marginalizing over the MSM of the transitions t_k

$$\mathrm{SO}_0^{N-1}(u_k) = \min_{t_k:\, u_k} [f_{k-1}(s_k) + M_k(t_k) + b_{k+1}(s_{k+1})] - \mathrm{SI}(u_k) \tag{10}$$

where u_k is either x_k or a_k. We refer to the operation in (10) as a *completion operation*.

While the forward–backward algorithm is computationally efficient, straightforward implementations of it have large latency [i.e., $\mathcal{O}(N)$] due to the ACS bottleneck in computing the causal and anticasual state MSMs.

B. Prefix and Suffix Operations

A prefix operation is defined as a generic form of computation that takes in n inputs $y_0, y_1, \ldots, y_{n-1}$ and produces n outputs $z_0, z_1, \ldots, z_{n-1}$ according to the following [13, Section 29.2.2], [14]:

$$z_0 = y_0 \tag{11}$$

$$z_i = y_0 \otimes \cdots \otimes y_i \tag{12}$$

where $\otimes$ is *any* associative binary operator.

Similarly, a suffix operation can be defined as a generic form of computation that takes in n inputs $y_0, y_1, \ldots, y_{n-1}$ and produces n outputs $z_0, z_1, \ldots, z_{n-1}$ according to

$$z_{n-1} = y_{n-1} \tag{13}$$

$$z_i = y_i \otimes \cdots \otimes y_{n-1} \tag{14}$$

[2]Note that, for generality, we allow the mapping from transitions to outputs to be dependent on k.

[3]In general K_1 and K_2 are functions of k. For notational compactness, we do not explicitly denote this dependency.

[4]As is the standard convention, the metric of a transition that cannot occur under the FSM structure is interpreted to be infinity.

[5]Any tail bit information can be enforced through the MI (b_k) terms in the tail. In most cases, this can also be achieved through nonuniform initialization of $b_N(s_N)$.

where $\otimes$ is *any* associative binary operator. Notice that a suffix operation is simply a (backward) prefix operation anchored at the other edge.

Prefix and suffix operations are important since they enable a class of algorithms that can be implemented with low latency using tree-structured architectures. The most notable realizations of this concept are VLSI N-bit tree adders with latency $\mathcal{O}(\lg N)$ [13], [20], [14].

III. Reformulation of the SISO Operation

The proposed low-latency architecture is derived by formulating the SISO computations in terms of a combination of prefix and suffix operations. To obtain this formulation, define $C(s_k, s_m)$, for $m > k$, as the MSM of state pairs s_k and s_m based on the soft-inputs between them, i.e., $C(s_k, s_m) = \mathrm{MSM}_k^{m-1}(s_k, s_m)$. The set of MSMs $C(s_k, s_m)$ can be considered an $(S \times S)$ matrix $\mathbf{C}(k, m)$. The causal state MSMs $\mathbf{f}_{k-1}$ can be obtained from $\mathbf{C}(0, k)$ by marginalizing (e.g., minimizing) out the condition on s_0. The backward state metrics can be obtained in a similar fashion. Specifically, for each conditional value of s_k

$$f_{k-1}(s_k) = \min_{s_0} C(s_0, s_k) \tag{15}$$

$$b_k(s_k) = \min_{s_N} C(s_k, s_N). \tag{16}$$

With this observation, the key step of the algorithm is to compute $\mathbf{C}(0, k)$ and $\mathbf{C}(k, N)$ for $k = 0, 1, \ldots, N-1$. Note that the inputs of the algorithm are the one-step transition metrics which can be written as $\mathbf{C}(k, k+1)$ for $k = 0, 1, \ldots, N-1$. To show how this algorithm can be implemented with a prefix and suffix computation, we define a min-sum fusion operator on $\mathbf{C}$ matrices that inputs two such matrices, one with a left-edge coinciding with the right-edge of the other, and marginalizes out the midpoint to obtain a pairwise state-MSM with larger span. Specifically, given $\mathbf{C}(k_0, m)$ and $\mathbf{C}(m, k_1)$, we define a $\mathbf{C}$ *fusion operator*, or $\otimes_C$ operator by

$$\mathbf{C}(k_0, k_1) \triangleq \mathbf{C}(k_0, m) \otimes_C \mathbf{C}(m, k_1) \iff$$
$$C(s_{k_0}, s_{k_1}) = \min_{s_m} [C(s_{k_0}, s_m) + C(s_m, s_{k_1})]$$
$$\forall\, s_{k_0}, s_{k_1}. \tag{17}$$

Note that the $\otimes_C$ operator is an associative binary operator that accepts two matrices and returns one matrix. This is illustrated in Fig. 1. With this definition $\mathbf{C}(0, k)$ and $\mathbf{C}(k, N)$ for $k = 0, 1, \ldots, N-1$ can be computed using the prefix and suffix operations as follows

$$\mathbf{C}(0, k) = \mathbf{C}(0, 1) \otimes_C \mathbf{C}(1, 2) \ldots \otimes_C \mathbf{C}(k-1, k)$$
$$\mathbf{C}(k, N) = \mathbf{C}(k, k+1) \otimes_C \ldots \otimes_C \mathbf{C}(N-2, N-1) \otimes_C \mathbf{C}(N-1, N).$$

In general, a SISO algorithm can be based on the decoupling property of state-conditioning. Specifically, conditioning on all possible FSM state values at time k, the shortest path problems (e.g., MSM computation) on either side of this state condition may be solved independently and then fused together (e.g., as performed by the $\mathbf{C}$-fusion operator). More generally, the SISO operation can be decoupled based on a partition of the observation interval with each subinterval processed independently and then fused together. For example, the forward–backward algorithm is based on a partition to the single-transition level with the fusing taking place *sequentially* in the forward and backward directions. In contrast, other SISO algorithms may be defined by specifying the partition and a schedule for fusing together the solutions to the sub-problems. This may be viewed as specifying an association scheme to the above prefix–suffix operations (i.e., grouping with parentheses).

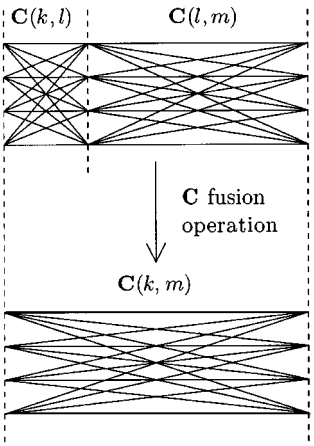

Fig. 1. $\mathbf{C}$ fusion operation.

The $\mathbf{C}$-fusion operations may be simplified in some cases depending on the association scheme. For example, the forward–backward algorithm replaces all $\mathbf{C}$-fusion operations by the much simpler forward and backward ACSs. However, latency is also a function of the association scheme. In the next section, we present an architecture based on a pairwise tree-structured grouping. This structure allows only a small subset of the $\mathbf{C}$-fusion operations to be simplified but facilitates a significant reduction in latency compared to the forward–backward algorithm by fusing solutions to the subproblems in a *parallel* instead of sequential manner.

IV. Low-Latency Tree-SISO Architectures

There are many known low-latency parallel architectures based on binary tree-structured groupings of prefix operations [20], [13], [14] that can be adopted to SISOs. All of these have targeted n-bit adder design where the binary associative operator is a simple one-bit addition. In fact, to the best of our knowledge, this is the first application of parallel prefix–suffix architectures to an algorithm based on binary associative operators that are substantially more complex than one-bit addition. The known parallel prefix architectures trade reduced area for higher latency and account for a secondary restriction of limited fanout of each computational module. This latter restriction is important when the computational modules are small and have delay comparable to the delay of wires and buffers (e.g., in adder design). The fusion operators, however, are relatively large. Consequently, given current VLSI trends, they will dominant the overall delay for the foreseeable future. Thus, we propose to adopt an architecture which minimizes

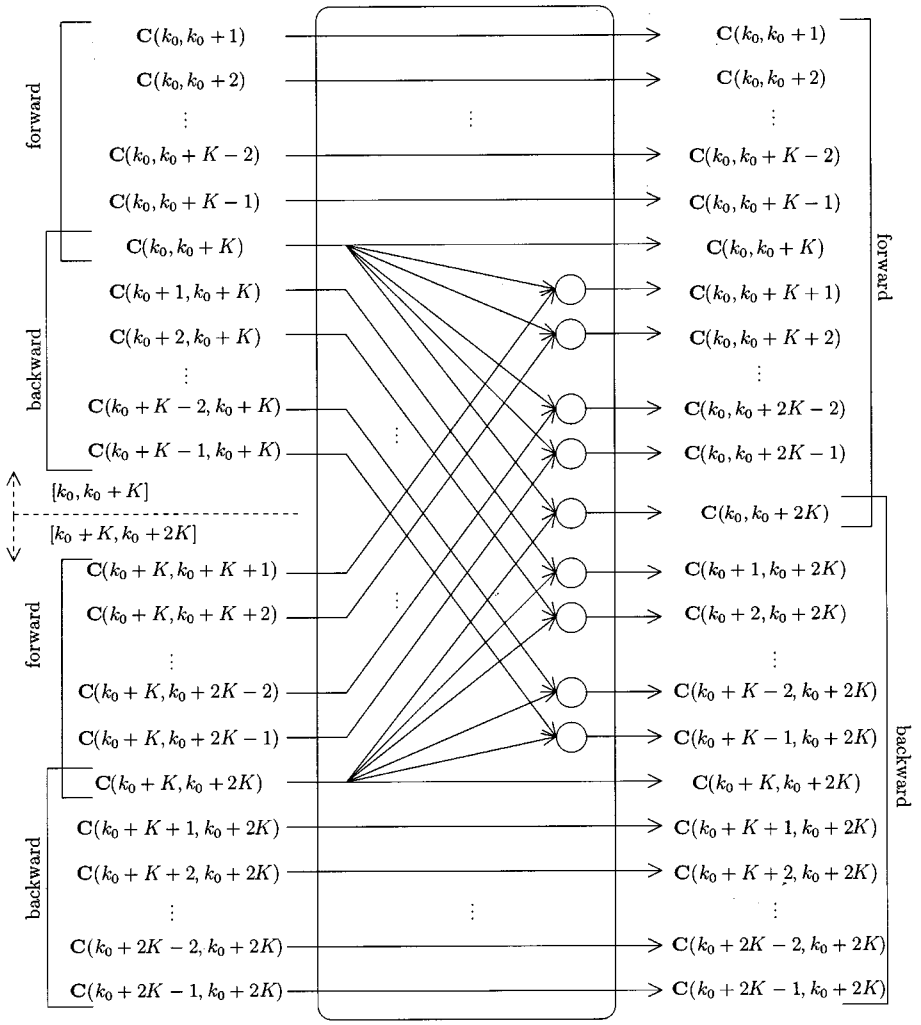

Fig. 2. Fusion module array for combining the complete set of $\mathbf{C}$ matrices on $[k_0,\ k_0+K]$ and $[k_0+K,\ k_0+2K]$ to obtain the complete set on $[k_0,\ k_0+2K]$.

latency with the minimal number of computational modules without regard to fanout [14].

Specifically, the forward and backward metrics, $\mathbf{f}_{k-1}$ and $\mathbf{b}_{N-k}$, for $k = 1, 2, \ldots, N$ can be obtained using a hierarchal tree-structure based on the *fusion-module* (FM) array shown in Fig. 2. We define a *complete set* of $\mathbf{C}$ matrices on the interval $\{k_0, k_0 + K\}$ as the $2K - 1$ matrices $\mathbf{C}(k_0, k_0 + m)$ and $\mathbf{C}(k_0 + m, k_0 + K)$ for $m = 1, 2, \ldots, K - 1$ along with $\mathbf{C}(k_0, k_0+K)$. This is the MSM information for all state pairs on the span of K steps in the trellis with one state being either on the left or right edge of the interval. The module in Fig. 2 fuses the complete sets of $\mathbf{C}$ matrices for two adjacent span-K intervals to produce a complete set of $\mathbf{C}$ matrices on the combined span of size $2K$. Of the $4K - 1$ output $\mathbf{C}$ matrices, $2K$ are obtained from the $2(2K-1)$ inputs without any processing. The other $2K-1$ output $\mathbf{C}$ matrices are obtained by $2K-1$ $\mathbf{C}$ *fusion modules*, or CFMs, which implement the $\otimes_C$ operator.

The basic span-K to span-$2K$ FM array shown in Fig. 2 can be utilized to compute the $\mathbf{C}$ matrices on the entire interval in $\lg N$ stages. This is illustrated in Fig. 3 for the special case of $N = 16$. Note that, indexing the stages from left to right (i.e., increasing span) as $i = 1, 2, \ldots, n = \lg N$ it is clear that there are 2^{n-i} FM arrays in stage i.

Because the final objective is to compute the causal and anticausal state metrics; however, not all FMs need be CFMs for all FM arrays. Specifically, the forward state metrics $\mathbf{f}_{k-1}$ can be obtained from $\mathbf{f}_{m-1}$ and $\mathbf{C}(m, k)$ via

$$f_{k-1}(s_k) = \min_{s_m} \left[f_{m-1}(s_m) + C(s_m, s_k) \right]. \tag{18}$$

Similarly, the backward state metrics can be updated via

$$b_k(s_k) = \min_{s_m} \left[b_m(s_m) + C(s_k, s_m) \right]. \tag{19}$$

We refer to a processing module that produces an $\mathbf{f}$ vector from another $\mathbf{f}$ vector and a $\mathbf{C}$ matrix, as described in (18), as an $\mathbf{f}$ *fusion module (fFM)*. A $\mathbf{b}$ fusion module (bFM) is defined analogously according to the operation in (19). In Fig. 3, we have indicated which FMs may be implemented as fFMs or bFMs.

The importance of this development is that the calculation of the state metrics has $\mathcal{O}(\lg N)$ latency. This is because the only data dependencies are from one stage to the next and thus all FM arrays within a stage and all FMs within an FM array can be executed in parallel, each taking $\mathcal{O}(1)$ latency. The cost of this low latency is the need for relatively large amounts of area. One mitigating factor is that, because the stages of the tree operate in sequence, hardware can be shared between stages. Thus,

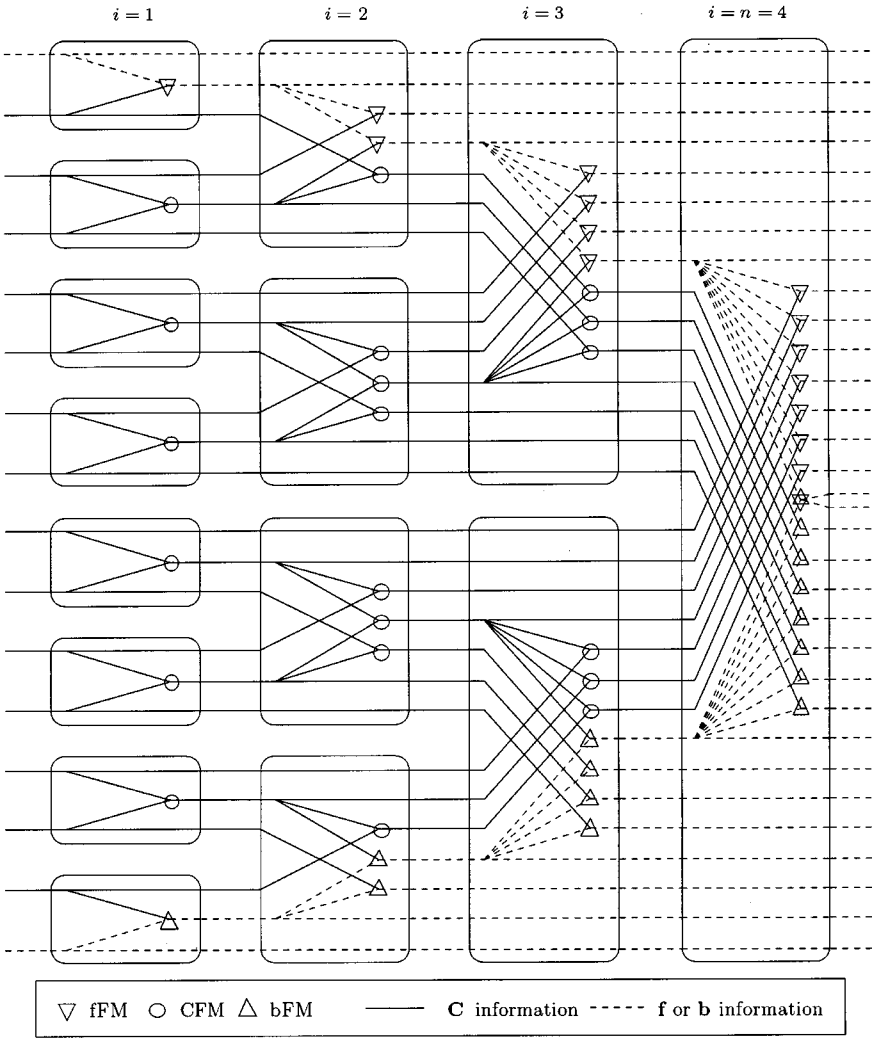

Fig. 3. Tree-SISO architecture for $N = 16$.

the stage that requires the most hardware dictates the total hardware needed. A rough estimate of this is N CFMs, each of which involves S^2 S-way ACS units with the associated registers. A more detailed analysis is given in Appendix II which accounts for the use of bFMs and fFMs whenever possible. For the example in Fig. 3 and a four-state FSM (i.e., $S = 4$), stage 2 has the most CFMs (8) but stage 3 has the most processing complexity. In particular, the complexity of stages $i = 1, 2, 3, 4$ measured in terms of four four-way ACS units is 26, 36, 32, and 16, respectively. Thus, if hardware is shared between stages, a total of 36 sets of four four-way ACS units is required to execute all FMs in a given stage in parallel. For applications when this number of ACS units is prohibitive, one can easily reduce the hardware requirements by as much as a factor of S with a corresponding linear increase in latency.

The implementation of the completion operation defined in (10) should also be considered. The basic operation required is a Q-way ACS unit where Q is the number of transitions consistent with u_k. Assuming that at most half of the transitions will be consistent with u_k, Q is upper bounded by $S^2/2$. Consequently, when S is large, low-latency, area-efficient implementations of the completion step may become an important issue. Fortunately, numerous low-latency implementations are well-known (e.g., [21]). The most straightforward may be one which uses a binary tree of comparators and has latency of $\mathcal{O}(\lg S^2)$. For small S, this additional latency is not significant.

The computational complexity of the state metric calculations can be computed using simple expressions based on Figs. 2 and 3. As shown in Appendix I, the total number of computations, measured in units of S S-way ACS computations

$$N_{S,\,S} = N((\lg N - 3)S + 2) + 4S - 2. \tag{20}$$

For the example in Fig. 3 and a four-state FSM, an equivalent of 110 sets of four four-way ACS operations are performed. This is to be compared with the corresponding forward–backward algorithm which would perform $2N = 32$ such operations and have baseline architectures with four times the latency. In general, note that the for a reduction in latency from N to $\lg N$, the computation is increased by a factor of roughly $(1/2)(\lg N - 3)S + 1$. Thus, while the associated complexity is high, the complexity scaling is sublinear in N. For small S, this is better than well-studied linear-scale solutions to low-latency Viterbi algorithm implementations (e.g., [21], [15]).

A. Optimizations for Sparse Trellises

The above architecture is most efficient for fully-connected trellises. For sparser trellis structures, however, the initial processing modules must process $\mathbf{C}$-matrices containing elements set to ∞, accounting for MSMs of pairs of states between which there is no sequence of transitions, thereby wasting processing power and latency. This section discusses optimizations that address this inefficiency.

For concreteness, we consider as a baseline a standard one-step trellis with $S = M^L$ states and exactly M transitions into and out of each state, in which, there exists exactly one sequence of transitions to go from a given state at time s_k to a given state s_{k+L}. One optimization is to precollapse the one-step trellis into an R-step trellis, $1 \leq R \leq L$, and apply the tree-SISO architecture to the collapsed trellis. A second optimization is to, wherever possible, simplify the $\mathbf{C}$ fusion modules. In particular, for a SISO on an R-step trellis, the first $\lg(L/R)$ stages can be simplified to banks of additions that simply add incoming pairs of multistep transition metrics.

More precisely, precollapsing involves adding the R metrics of the one-step transitions that constitute the transition metrics of each *supertransition* $\mathbf{t}_{kR}^{(k+1)R}$, for $k = 0, 1, \ldots, (N-1)/R$. The SISO accepts these inputs and produces forward and backward MSMs, $f_{kR-1}(s_k)$ and $b_{(k+1)R}(s_{(k+1)R})$, for $k = 0, 1, \ldots, N/R$. The key benefit of precollapsing is that the number of SISO inputs is reduced by a factor of R, thereby reducing the number of stages required in the state metric computation by $\lg R$. One disadvantage of precollapsing is that the desired soft-outputs must be computed using a more complex, generalized completion operation. Namely

$$\begin{aligned}\mathrm{SO}_0^{NR-1}(u_{kR+m}) &= \min_{\mathbf{t}_{kR}^{(k+1)R}:\, u_{kR+m}} \Big[f_{kR-1}(s_k) + M_{kR}^{(k+1)R}\left(\mathbf{t}_{kR}^{(k+1)R}\right) \\ &\qquad + b_{(k+1)R+1}\left(s_{(k+1)R}\right) \Big] - \mathrm{SI}(u_{kR+m}) \\ &\quad m = 0, 1, \ldots, R-1. \end{aligned} \tag{21}$$

The principle issue is that for each u_{kR+m} this completion step involves an $(M^{L+R}/2)$-way ACS rather than the $(M^{L+1}/2)$-way ACS required for the one-step trellis.

In order to identify the optimal R (i.e., for minimum latency) assuming both these optimizations are performed, the relative latencies of the constituent operations are needed. While exact latencies are dependent on implementation details, rough estimates may still yield insightful results. In particular, we can assume that both the precollapsing additions and ACS operations for the state metric and completion operations are implemented using binary trees of adders/comparators and, therefore, estimate that their delay is logarithmic in the number of their inputs. An important observation is that the precollapsing along with $\lg R$ simplified stages together add L one-step transition metrics (producing the transition metrics for a fully-connected L-step trellis) and thus can jointly be implemented in an estimated $\lg L$ time units. In addition, the state metric (M^L)-way ACS units take $\lg M^L$ time units and the completion units $(M^{L+R}/2)$-way ACSs take $\lg(M^{L+R}/2)$ time units. Assuming maximal parallelism, this yields a total latency of

$$\lg L + \lg(N/R)\lg(M^L) + \lg(M^{L+R}/2). \tag{22}$$

It follows that the minimum latency occurs when $R - L\lg R$ is minimum (subject to $1 \leq R \leq L$), which occurs when $R = L$. This suggests that the minimum-latency architecture is one in which the trellis is precollapsed into a fully-connected trellis and more complex completion units are used to extract the soft outputs from the periodic state metrics calculated.

The cost of this reduced latency is the additional area required to implement the trees of adders that produce the L-step transition metrics and the larger trees of comparators required to implement the more complex completion operations. Note, however, that this area overhead can be mitigated by sharing adders and comparators among stages of each tree and, in some cases, between trees with only marginal impact on latency.

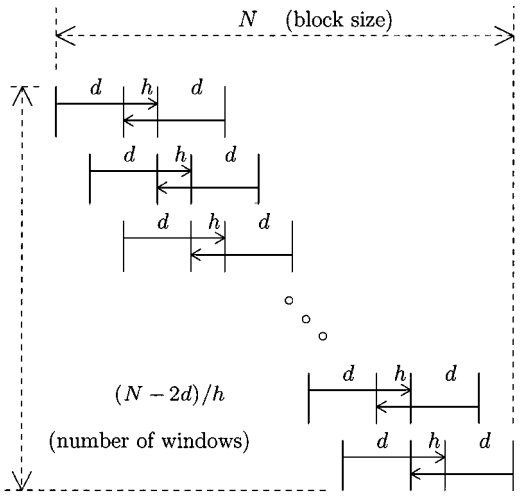

Fig. 4. Tiled subwindow scheme based on the forward–backward algorithm.

V. Use in Tiled Subwindow Schemes

One known method of reducing latency and improving throughput of computing the soft-inverse is to use smaller combining windows.[6] We define *minimum half-window (MHW)* algorithms as those in which the combing window edges K_1 and K_2 satisfy $K_1 \leq \max(0, k-d)$ and $K_2 \geq \min(N, k+d)$, for $k = 0, \ldots, N-1$—i.e., for every point k away from the edge of the observation window, the soft-output is based on a subwindow with left and right edges at least d points from k.

The traditional forward–backward algorithm can be used on subwindows to obtain a MHW-SISO. One particular scheme is the *tiled subwindow technique* in which combining windows of length $2d+h$ are used to derive all state metrics. In this scheme, as illustrated in Fig. 4, the windows are tiled with overlap of length $2d$ and there are $(N-2d)/h$ such windows. The forward–backward recursion on each interior subwindow yields h

[6]We emphasize that when only a partial combining window is used, the actual soft-inverse is not computed. However, for sufficiently large combining windows, the soft-inverse should be well-approximated.

soft outputs, so there is an *overlap penalty* which increases as h decreases.

For the ith such window, the forward and backward state metrics are computed using the recursions, modified from that of (7) and (9)

$$f_k^{(i)}(s_{k+1}) \triangleq \text{MSM}_{ih}^{k}(s_{k+1}) \tag{23}$$

$$b_k^{(i)}(s_k) \triangleq \text{MSM}_{k}^{ih+2d+h-1}(s_k). \tag{24}$$

If all windows are processed in parallel, this architecture yields a latency of $\mathcal{O}(d+h)$.

The tree-SISO algorithm can be used in a MHW scheme without any overlap penalty and with $\mathcal{O}(\lg\, d)$ latency. Consider N/d combining windows of size d and let the tree-SISO compute $C(id,\, id+j)$ and $C((i+1)d,\, (i+1)d-j)$ for $j=0,\, \ldots,\, d-1$ and $i=0,\, \ldots,\, N/d-1$. Then, use one additional stage of logic to compute the forward and backward state metrics for all k time indices that fall within the ith window, $i=0,\, \ldots,\, N/d-1$, as follows:[7]

$$\begin{aligned} &f_k^{(i)}(s_{k+1}) \\ &\quad \triangleq \text{MSM}_{(i-1)d}^{k}(s_{k+1}) \\ &\quad = \min_{s_{id}} \left\{ \left[\min_{s_{(i-1)d}} C\left(s_{(i-1)\,d},\, s_{id}\right) \right] + C(s_{id},\, s_{k+1}) \right\} \end{aligned} \tag{25}$$

$$\begin{aligned} &b_k^{(i)}(s_k) \\ &\quad \triangleq \text{MSM}_{k}^{(i+1)d}(s_k) \\ &\quad = \min_{s_{id}} \left\{ C(s_k,\, s_{id}) + \left[\min_{s_{(i+1)d}} C\left(s_{id},\, s_{(i+1)\,d}\right) \right] \right\}. \end{aligned} \tag{26}$$

The inner minimization corresponds to a conversion from $\mathbf{C}$ information to $\mathbf{f}$ ($\mathbf{b}$) information as in (15) and (16). The outer minimization corresponds to an fFM or bFM. The order of this minimization was chosen to minimize complexity. This is reflected in the example of this approach shown in Fig. 5, where the last stage of each of the four tree-SISOs is modified to execute the above minimizations in the proposed order. We refer to the module that does this as a *2Cfb module*. This module may be viewed as a specialization of the stage 2 center CFMs in Fig. 3. The above combining of subwindow tree-SISO outputs adds one additional processing stage so that the required number of stages of FMs is $\lg(d)+1$.

A. Computational Complexity Comparison

The computational complexity of computing the state metrics using the forward–backward tiled scheme is the number of windows times the complexity of computing the forward–backward algorithm on each window. In terms of S S-way ACSs, this can be approximated for large N via

$$\frac{N-2d}{h}\, 2(d+h) \approx \frac{2N}{h}\,(d+h). \tag{27}$$

The computational complexity of computing the state metrics using the tree-SISO tiled scheme in terms of S S-way ACSs can be developed similarly and is

$$\frac{N}{d}\, d\lg(d)S + 2N = N(S\lg(d)+2). \tag{28}$$

Determining which scheme has higher computational complexity depends on the relative sizes of h and d. If h is reduced, the standard forward–backward scheme reduces in latency but increases in computational complexity because the number of overlapped windows increase. Since the tiled tree-SISO architecture has no overlap penalty, as h is decreased in a tiled forward–backward scheme, the relative computational complexity trade-off becomes more favorable to the tree-SISO approach. In fact, for $h < 2d/S\lg d$, the computational complexities of the tree-SISO is lower that the tiled forward–backward scheme.

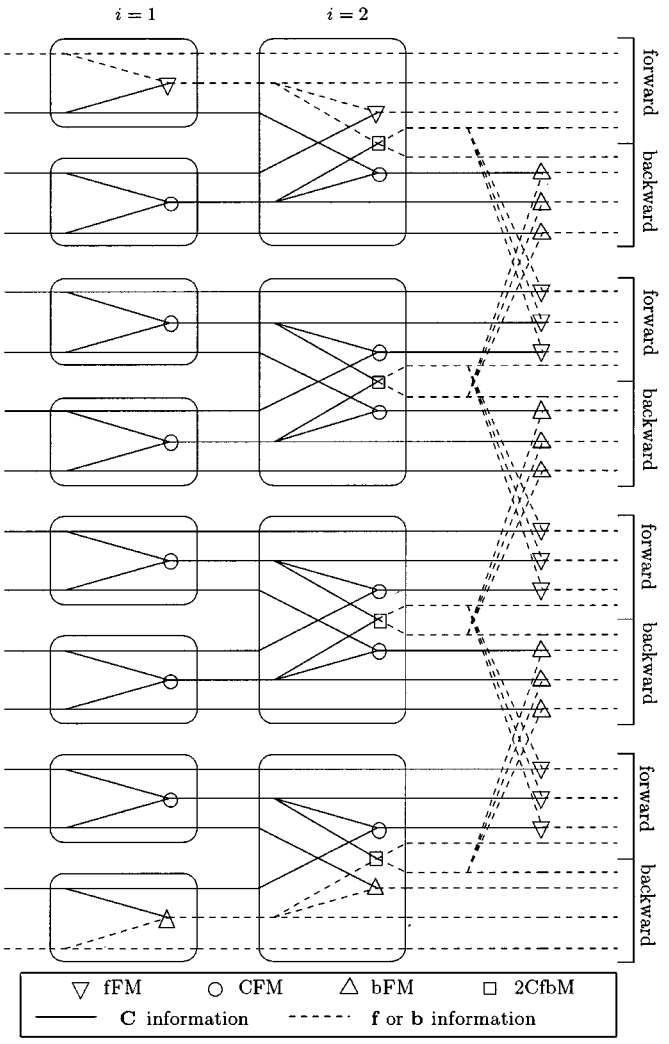

Fig. 5. Tiled subwindow approach with four tree-SISOs of window size 4 for $N=16$ to implement a $d=4$ MHW SISO.

[7]This should be interpreted with $C(s_{-d},\, s_0)$ replaced by initial left-edge information and similarly for $C(s_{N-1},\, s_{N+d-1})$.

VI. Design Example: Four-State PCCC

The highly parallel architectures considered require large implementation area. In this section, we consider an example for which the area requirements are most feasible for implementation in the near future. Specifically, we consider an iterative decoder based on four-state sparse (one-step) trellises. Considering larger S will yield more impressive latency reductions for the tree-SISO. This is because the latency-reduction obtained by the tree-SISO architecture relative to the parallel tiled forward–backward architecture depends on the minimum half-window size. One expects that good performance requires a value of d that grows with the number of states (i.e., similar to the rule-of-thumb for traceback depth in the Viterbi algorithm [22] for sparse trellises). In contrast, considering precollapsing will yield less impressive latency reductions. For example, if $d = 16$ is required for a single-step trellis, then an effective value of $d = 8$ would suffice for a two-step trellis. The latency reduction factor associated with the tree-SISO for the former would be approximately four but only 8/3 for the latter. However, larger S and/or precollapsing yields larger implementation area and is not in keeping with our desire to realistically assess the near-term feasibility of these algorithms.

In particular, we consider a standard parallel concatenated convolutional code (PCCC) with two four-state constituent codes [1], [2]. Each of the recursive systematic constituent codes generates parity using the generator polynomial $G(D) = (1 + D^2)/(1 + D + D^2)$ with parity bits punctured to achieve an overall systematic code with rate 1/2.

In order to determine the appropriate value for d to be used in the MHW-SISOs, we ran simulations where each SISO used a combining window $\{k - d, \ldots, k + d\}$ to compute the soft-output at time k. This is exactly equivalent to the SISO operation obtained by a tiled forward–backward approach with $h = 1$. Note that, since d is the size of all (interior) half-windows for the simulations, any architecture based on a MHW-SISO with d will perform at least as well (e.g., $h = 2$ tiled forward–backward, d-tiled tree-SISO, etc.). Simulation results are shown in Fig. 6 for an interleaver size of $N = 1024$ with min-sum marginalization and combining and ten iterations. The performance is shown for various d along with the performance of the fixed-interval ($N = 1024$) SISO. No significant iteration gain is achieved beyond ten iterations for any of the configurations. The results indicate that $d = 16$ yields performance near the fixed-interval case. This is consistent with the rule-of-thumb of five to seven times the memory for the traceback depth in a Viterbi decoder (i.e., roughly $d = 7 \times 2 = 14$ is expected to be sufficient).

Since the required window size is $d = 16$, the latency improvement of a tree-SISO relative to a tiled forward–backward scheme is close to $4 = 16/\lg(16)$. The computational complexity of these two approaches is similar and depends on the details of the implementation and the choice of h for the tiled forward–backward approach. A complete fair comparison would require a detailed implementation of the two approaches. Below, we summarize a design for the tree-SISO based subwindow architecture.

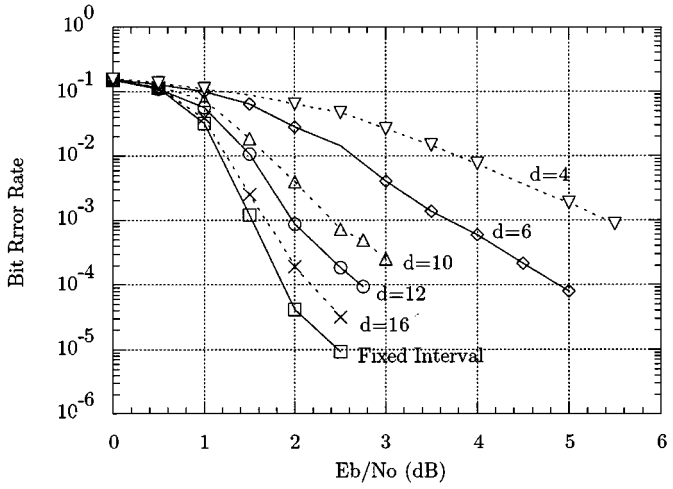

Fig. 6. Simulation results for a standard turbo code decoded using SISOs with various half-window sizes, $N = 1024$, and ten iterations.

A factor that impacts the area of the architecture is the bit-width of the data units. Simulation results suggest that an eight-bit datapath is sufficient. Roughly speaking, a tree-based architecture for this example would require 1024 sets of sixteen four-way ACS units along with associated output registers to store intermediate state metric results. Each four-way ACS unit can be implemented with an eight-bit 4:1 multiplexor, four eight-bit adders, six eight-bit comparators, and one eight-bit register [21]. Our initial VLSI designs indicate that these units require approximately 2250 transistors. Thus, this yields an estimate of $16 \times 2250 \times 1024 \approx 40$ Million transistors. This number or logic transistors pushes the limit of current VLSI technology but should soon be feasible. We consider an architecture in which one clock cycle is used per stage of the tree at a 200 MHz clock frequency. For $d = 16$, each SISO operation can be performed in six such clock cycles (using one clock for the completion step). Moreover, we assume a hard-wired interleaver comprising two rows of 1024 registers with interconnection an network. Such an interleaver would be larger than existing memory-based solutions [10] but could have a latency of one clock cycle. Consequently, one iteration of the turbo decoder, consisting of two applications of the SISO, one interleaving, and one deinterleaving, requires 14 clock cycles. Assuming ten iterations, the decoding of 1024 bits would take 140 clock cycles, or a latency of just 700 ns.

This latency also implies a very high throughput which can further be improved with standard pipelining techniques. In particular, a nonpipelined implementation has an estimated throughput of 1024 bits per 700 ns = 1.5 Gb/s. Using the tree-SISO architecture, one could also pipeline across interleaver blocks as described by Masera *et al.* [10]. In particular, 20 such tiled tree-SISOs and associated interleavers can be used to achieve a factor of 20 in increased throughput, yielding a throughput of 30 Gb/s.

Moreover, unlike architectures based on the forward–backward algorithm, the tree-SISO can easily be internally pipelined, yielding even higher throughputs with linear hardware scaling. In particular, if dedicated hardware is used for each stage of

the tree-SISO, pipelining the tree-SISO internally may yield another factor of $\lg(d)$ in throughput, with no increase in latency. For window sizes of $d = 16$, the tree-based architecture could support over 120 Gb/s. That said, it is important to realize that with current technology such hardware costs may be beyond practical limits. Given the continued increasing densities of VLSI technology, however, even such aggressive architectures may become cost-effective in the future.

VII. Conclusion

Based on the interpretation of the SISO operation in terms of parallel prefix/suffix operations, a family of tree-structured architectures were suggested. Compared to the baseline forward– backward algorithm architecture, the tree-SISO architecture reduces latency from $\mathcal{O}(N)$ to $\mathcal{O}(\lg N)$. More recently, alternative tree-structured SISOs have been developed that trade a linear increase in latency for substantially lower complexity and area. In particular, other existing architectures for parallel prefix/suffix computations from the VLSI literature have been applied to the SISO computation [23]. Also, it has been demonstrated that tree-structured SISOs can be derived as an application of message-passing on a binary tree model for the FSM [24], [18].

An efficient SISO design may not be built using a single tree-SISO but rather using tree-SISOs as important components. For example, in this paper, many tree-SISOs were used to comprise a SISO using tiled subwindows. Latency in this case is reduced from linear in the minimum half-window size (d) for fully-parallel tiled architectures based on the forward–backward algorithm, to logarithmic in d for tiled tree-SISOs. More recently, a high-radix SISO (e.g., 16 steps) was designed using a tree-SISO to compute the multistep metrics with low latency [23].

In general, the potential latency advantages of the tree-SISO are clearly most significant for applications requiring large combining windows. For most practical designs, this is expected when the number of states increases. In the one detailed four-state tiled-window example considered, the latency was reduced by a factor of approximately four. For systems with binary inputs and S states, one would expect that $d \cong 8\lg(S)$ would be sufficient. Thus, there is a potential reduction in latency of approximately $8\lg(S)/\lg(8\lg S)$ which becomes quite significant as S increases. However, the major challenge in achieving this potential latency improvement is the area required for the implementation. In particular, building a high-speed S-way ACS unit for large S is the key challenge. Techniques to reduce this area requirement without incurring performance degradations (e.g., bit-serial architectures) are promising areas of research. In fact, facilitating larger S may allow the use of smaller interleavers which alleviates the area requirements and reduces latency.

Appendix I
Computation Complexity Analysis

The number of required stages is $n = \lg N$, with 2^{n-i} FM arrays in stage i. Each of these FM arrays in stage i span 2^i steps in the trellis and contains $2^i - 1$ FMs. Thus, the total number of FMs in stage i is $n_{\mathrm{FM}}(i) = (2^i - 1)2^{n-i}$. The total number of fusion operations is therefore

$$\begin{aligned} N_{\mathrm{FM}} &= \sum_{i=1}^{n} n_{\mathrm{FM}}(i) \\ &= \sum_{i=1}^{n} 2^n - 2^n 2^{-i} \\ &= Nn - N\sum_{i=1}^{n} 2^{-i} \\ &= N(\lg N - 1) + 1. \end{aligned} \tag{29}$$

For the example, in Fig. 3, this reduces to $N_{\mathrm{FM}} = 49$.

Using Fig. 3 as an example, it can be seen that the number of FMs that can be implemented as fFMs in stage i is $n_{\mathbf{f}}(i) = 2^{i-1}$. In the special case of $i = n$, this must be interpreted as replacing the $2K - 1$ CFMs by K fFMs and K bFMs. For example, in the fourth stage in Fig. 3, the 15 CFMs implied by Fig. 2 may be replaced by eight fFMs and eight bFMs, as shown. The number of FMs that can implemented as bFMs is the same—i.e., $n_{\mathbf{b}}(i) = n_{\mathbf{f}}(i) = 2^{i-1}$. It follows that the number of CFMs required at stage i is

$$n_{\mathbf{C}}(i) = n_{\mathrm{FM}}(i) - n_{\mathbf{b}}(i) - n_{\mathbf{f}}(i) + \delta(n-i) \tag{30}$$

$$= 2^n - 2^{n-i} - 2^i + \delta(n-i) \tag{31}$$

where $\delta(j)$ is the Kronecker delta. The total number of fusion modules is therefore

$$N_{\mathbf{f}} = N_{\mathbf{b}} = \sum_{i=1}^{n} n_{\mathbf{f}}(i) = \sum_{i=1}^{n} 2^{i-1} = N - 1 \tag{32}$$

$$\begin{aligned} N_{\mathbf{C}} &= \sum_{i=1}^{n} n_{\mathbf{C}}(i) \\ &= N(n-1) + 1 - \left(\sum_{i=1}^{n} 2^i\right) + 1 \\ &= N(\lg N - 3) + 4. \end{aligned} \tag{33}$$

Comparing (29) and (33), it is seen that, for relatively large N, the fraction of FMs that must be CFMs is $(\lg N - 3)/(\lg N - 1)$. For smaller N, the fraction is slightly larger. For example, in Fig. 3, $N_{\mathbf{f}} = N_{\mathbf{b}} = 15$ and there are 20 CFMs.

The CFM is approximately S (i.e., the number of states) times more complex than the fFM and bFM operations. This can be seen by comparing (17) with (18) and (19). Specifically, the operations in (17)–(19) involve S-way ACSs. For the CFM, an S-way ACS must be carried out for every possible state pair (s_{k_0}, s_{k_1}) in (17)—i.e., S^2 state pairs. The S-way ACS operations in (18), and (19) need only be computed for each of the S states s_k. Thus, taking the basic unit of computation to be S

S-way ACS on an S-state trellis, the total number of these computations required for stage i is

$$n_{S,S}(i) = Sn_{\mathbf{C}}(i) + n_{\mathbf{f}}(i) + n_{\mathbf{b}}(i). \tag{34}$$

Summing over stages, we obtain the total number of computations, measured in units of S S-way ACS computations

$$N_{S,S} = SN_{\mathbf{C}} + 2N_{\mathbf{f}} = N((\lg N - 3)S + 2) + 4S - 2 \tag{35}$$

which is restated in (20).

Appendix II
Hardware Resource Requirements

The maximum of $n_{S,S}(i)$ over i is of interest because it determines the minimum hardware resource requirements to achieve the desired minimum latency. This is because the fusion modules can be shared between stages with negligible impact on latency.

The maximum of $n_{S,S}(i)$ can be found by considering the condition on i for which $n_{S,S}(i) \geq n_{S,S}(i-1)$. Specifically, if $i < n$

$$n_{S,S}(i) \geq n_{S,S}(i-1) \tag{36}$$

$$\iff \quad 2^n \geq 2^{2i-1}(1 - S^{-1}) \tag{37}$$

$$\iff \quad i \leq \frac{n + 1 - \lg(1 - S^{-1})}{2}. \tag{38}$$

It follows that $n_{S,S}(i)$ has no local maxima and

$$i^* = \left\lfloor \frac{n + 1 - \lg(1 - S^{-1})}{2} \right\rfloor \tag{39}$$

can be used to find the maximizer of $n_{S,S}(i)$. Specifically, if (39) yields $i^* < n-1$, then the maximum occurs at i^*, otherwise ($i^* = n-1$), the $i = n-1$ and $i = n$ cases should be compared to determine the maximum complexity stage.[8] For $S \geq 4$, (39) can be reduced to

$$i^* = \left\lfloor \frac{n+1}{2} \right\rfloor \tag{40}$$

since $0.5 \leq (1 - \lg(1 - S^{-1}))/2 \leq 0.71$ for $S \geq 4$.

Acknowledgment

The authors thank an anonymous reviewer for carefully reading this paper and providing helpful comments. The authors also thank P. Thiennviboon for his helpful comments.

[8]For example, with $n = 2$ and four states, (39) yields $i^* = 1$ but $n_{S,S}(2) > n_{S,S}(1)$.

References

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

[2] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Trans. Commun.*, vol. 44, pp. 1261–1271, Oct. 1996.

[3] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-input soft-output modules for the construction and distributed iterative decoding of code networks," *Eur. Trans. Telecommun.*, vol. 9, no. 2, pp. 155–172, Mar. 1998.

[4] X. Chen and K. M. Chugg, "Near-optimal page detection for two-dimensional ISI/AWGN channels using concatenated modeling and iterative detection," in *Proc. Int. Conf. Communications*, vol. 2, Atlanta, GA, June 1998, pp. 952–956.

[5] K. M. Chugg, X. Chen, A. Ortega, and C.-W. Chang, "An iterative algorithm for two-dimensional digital least metric problems with applications to digital image compression," in *Proc. Int. Conf. Image Processing*, vol. 2, Chicago, IL, Oct. 1998, pp. 722–726.

[6] M. Moher and P. Guinand, "An iterative algorithm for asynchronous coded multiuser detection," *IEEE Commun. Lett.*, vol. 2, pp. 229–231, Aug. 1998.

[7] X. Chen and K. M. Chugg, "Reduced state soft-in/soft-out for complexity reduction in iterative and noniterative data detection," in *Proc. Int. Conf. Communications*, vol. 1, New Orleans, LA, June 2000, pp. 6–10.

[8] G. D. Forney Jr., "The Viterbi algorithm," *Proc. IEEE*, vol. 61, pp. 268–278, Mar. 1973.

[9] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

[10] G. Masera, G. Piccinini, M. Ruo Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. VLSI Syst.*, vol. 7, Sept. 1999.

[11] A. Bishop, I. Chan, S. Aronson, P. Moran, K. Hen, R. Cehng, K. K. Fitzpatrick, J. Stander, R. Chik, K. Kshonze, M. Aliahmad, J. Ngai, H. He, E. daVeiga, P. Bolte, C. Krasuk, B. Cerqua, and R. Brown, "A 300 Mb/s BiCMOS disk drive channel with adaptive analog equalizer," in *Proc. IEEE Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1999, pp. 46–49.

[12] W. Ryan, "Performance of high rate turbo codes on a pr4-equalized magnetic recording channel," in *Proc. Int. Conf. Communications*, vol. 2, Atlanta, GA, June 1998, pp. 947–951.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: M.I.T. Press, 1990.

[14] A. E. Despain, "Chapter 2: Notes on computer architecture for high performance," in *New Computer Architectures*. New York: Academic, 1984.

[15] G. Fettweis and H. Meyr, "Parallel Viterbi algorithm implementation: Breaking the ACS-bottleneck," *IEEE Trans. Commun.*, vol. 37, pp. 785–790, Aug. 1989.

[16] N. Wiberg, "Codes and Decoding on General Graphs," Ph.D. dissertation, Dept. of Electrical Engineering, Linköping University, Linköping, Sweden, 1996.

[17] S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Trans. Inform. Theory*, vol. 46, pp. 325–343, Mar. 2000.

[18] K. M. Chugg, A. Anastasopoulos, and X. Chen, *Iterative Detection: Adaptivity, Complexity Reduction, and Applications*. Norwell, MA: Kluwer, 2001.

[19] P. Robertson, E. Villebrum, and P. Hoeher, "A comparison of optimal and suboptimal MAP decoding algorithms operating in the log domain," in *Proc. Int. Conf. Communications*, Seattle, WA, June 1995, pp. 1009–1013.

[20] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, pp. 260–264, Mar. 1982.

[21] P. J. Black, "Algorithms and Architecturees for High-Speed Viterbi Decoding," Ph.D. dissertation, Electrical Engineering Dept., Stanford University, Stanford, CA, 1993.

[22] J. A. Heller and I. M. Jacobs, "Viterbi decoding for satellite and space communication," *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 835–848, Oct. 1971.

[23] G. D. Dimou, "Low latency turbo decoders," M.S. thesis, Electrical Engineering - Systems Dept., University of Southern California, Los Angeles, CA, Dec. 2000.

[24] P. Thiennviboon and K. M. Chugg, "A low-latency SISO via message passing on a binary tree," presented at the Allerton Conf. Commun., Control, Comp., Allerton Park, IL, Oct. 2000.

Peter A. Beerel (S'88–M'95) received the B.S.E. degree in electrical engineering from Princeton University, Princeton, NJ, in 1989, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1991 and 1994, respectively.

He joined the Department of Electrical Engineering—Systems at University of Southern California (USC), Los Angeles, in 1994, where he is currently an Associate Professor. His research interests include a variety of topics in CAD and VLSI. He has consulted for Yuni Networks and AMCC in the areas of networking chip design, Intel and Asynchronous Digital Design in the areas of asynchronous design and CAD, and TrellisWare Technologies in the area of communication chip design.

Dr. Beerel was a recipient of an Outstanding Teaching Award in 1997 and the Junior Research Award in 1998, both from USC's School of Engineering. He received a National Science Foundation (NSF) Career Award and a 1995 Zumberge Fellowship. He was also co-winner of the Charles E. Molnar Award for two papers published in ASYNC'97 that best bridged theory and practice of asynchronous system design and was a co-recipient of the best paper award in ASYNC'99. Dr. Beerel is co-author of four patents in the area of asynchronous circuits. He has been a Member of the Technical Program Committee for the International Symposium on Advanced Research in Asynchronous Circuits and Systems since 1997 and was Program Co-chair for ASYNC'98. He has served on the Technical Program Committee for ICCAD'00 and is currently serving on the Technical Program Committee of ICCAD'01.

Keith M. Chugg (S'88–M'95) received the B.S. degree (with high distinction) in engineering from Harvey Mudd College, Claremont, CA, in 1989 and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California (USC), Los Angeles, in 1990 and 1995, respectively.

During the 1995–1996 academic year, he was an Assistant Professor with the Electrical and Computer Engineering Department, University of Arizona, Tucson. In 1996, he joined the EE-Systems Dept., USC, where he is currently an Associate Professor. His research interests are in the general areas of signaling, detection, and estimation for digital communication and data storage systems. He is also interested in architectures for efficient implementation of the resulting algorithms. Along with his former Ph.D. students, A. Anastasopoulos and X. Chen, he is co-author of the book *Iterative Detection: Adaptivity, Complexity Reduction, and Applications* (Kluwer: Norwell, MA). He is a Co-Founder of TrellisWare Technologies, Inc., a company dedicated to providing complete likelihood-based digital baseband solutions, where he serves as a Consultant and member of the Technical Advisory Board.

Dr. Chugg currently is serving as an Editor for the IEEE TRANSACTIONS ON COMMUNICATIONS in the area of Signal Processing and Iterative Detection and as Technical Program Chair for the Communication Theory Symposium at Globecom 2002.