VLSI Architectures for Iterative Decoders in Magnetic Recording Channels

Engling Yeo, *Student Member, IEEE*, Payam Pakzad, Borivoje Nikolić, *Member, IEEE,* and Venkat Anantharam, *Fellow, IEEE*

***Abstract*—VLSI implementation complexities of soft-input soft-output (SISO) decoders are discussed. These decoders are used in iterative algorithms based on Turbo codes or Low Density Parity Check (LDPC) codes, and promise significant bit error performance advantage over conventionally used partial-response maximum likelihood (PRML) systems, at the expense of increased complexity. This paper analyzes the requirements for computational hardware and memory, and provides suggestions for reduced-complexity decoding and reduced control logic. Serial concatenation of interleaved codes, using an outer block code with a partial response channel acting as an inner encoder, is of special interest for magnetic storage applications.**

***Index Terms*—Iterative decoders, LDPC codes, magnetic recording, turbo codes, VLSI architectures.**

I. Introduction

A TURBO encoder using serial concatenation of a convolutional code or Low Density Parity Check (LDPC) code with a partial-response channel acting as the inner coder is shown in Fig. 1 [1]. The iterative decoder (Fig. 2) uses a combination of soft-input-soft-output (SISO) decoders separated by interleavers, π, and the inverse, π^{-1}. We present SISO decoder implementations that employ either the MAP Algorithm (BCJR) [2], Soft Output Viterbi Algorithm (SOVA) [3], or the LDPC decoding algorithm [5].

All systems considered in this paper assume an E^2PR4 partial response channel. The particular partial response target is not essential to the following discussion, and E^2PR4 is used as an example because it presents a complexity equivalent to contemporary read channel detectors. The outer code is either a 16-state binary convolutional code or an LDPC code, implementing a rate 8/9 coding. As is common with most magnetic recording channels, the use of block codes and interleaver design is restricted to a sector size of 4096 user bits. The number of bits used to represent the log-likelihood ratios or messages is a tradeoff between the amount of hardware required and the BER performance of the iterative decoder. Earlier systems using 4 to 6-bit representations [6], [7] have reported good performance with respect to floating-point results.

Manuscript received June 20, 2000; revised Oct 11, 2000. The work of P. Pakzad and V. Anantharam was supported by the National Science Foundation through Awards IRI-97-12131 and SBR-9873086, and by ONR MURI through Award N00014-1-0637 on "Decision Making Under Uncertainty."

The authors are with the Electrical Engineering and Computer Sciences Department, University of California, Berkeley, CA 94720 USA (e-mail: yeo@eecs.berkeley.edu).

Publisher Item Identifier S 0018-9464(01)02388-3.

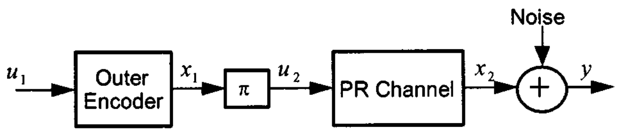

Fig. 1. Serially concatenated turbo encoder with a convolutional outer code.

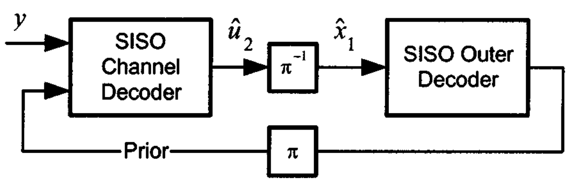

Fig. 2. Iterative decoder using SISO decoders separated by interleavers.

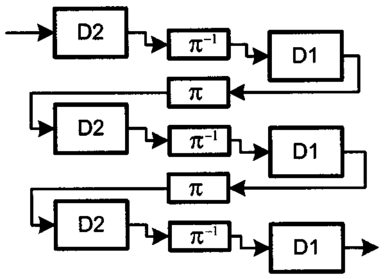

Fig. 3. Pipelined decoder for serially concatenated turbo codes using outer decoder D1 and inner decoder D2 separated by interleavers/deinterleavers, π/π^{-1}.

In order to achieve desired throughputs (above 1 Gbps) that are in line with current trends in magnetic recording systems, a fully unrolled and pipelined architecture [6] is needed (Fig. 3). This results in a linear complexity increase with the number of iterations.

In the following sections, structures for the building blocks of an iterative decoder will be analyzed. Section II discusses the implementation of an interleaver and deinterleaver. Section III discusses a MAP decoder implementing the Windowed-BCJR algorithm, using a minimal number of Add–Compare–Select units and a highly regular memory access pattern. A realization of a SOVA decoder by a simple extension of the register exchange method is presented in Section IV. Section V discusses a pipelined LDPC decoder and proposes a message arrangement in memory that lowers the complexity for address decoding. Section VI compares the results and Section VII provides some concluding remarks.

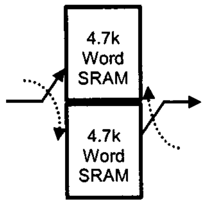

Fig. 4. Interleavers and deinterleavers implemented using alternating read/write buffers.

II. Interleaver

The randomness of the interleaver output sequence makes it difficult to realize in-place storage. A direct interleaver implementation uses two banks of buffers alternating between read/write for consecutive sectors of data (Fig. 4). The latency through an interleaver is therefore equal to the block size.

The basic block interleaver design uses a minimal amount of control logic. Using static random-access memory (SRAM) for high-speed implementation, the interleaver inputs are written row-wise in the memory array, while outputs are read column-wise. For a block interleaver of size N arranged as an X by Y matrix, such that $N = XY$, this assures that bits located within a distance of $Y - 1$ before interleaving are separated by a minimum distance of X after interleaving. The sequential write/read pattern along rows/columns allows the memory access operations of this interleaver to make use of cycle counters to activate both word (row) lines and bit (column) lines, thereby eliminating the necessity to perform memory-address decoding.

More sophisticated interleaver designs [8], [9] yield improved error rate performance, but result in increased implementation complexity. Therefore, the implementation of the described basic interleaver provides a lower limit on complexity.

III. MAP Decoder

A MAP decoder implements the BCJR [2] algorithm. It is used to obtain the *a posteriori* information for partial response channel decoding, as well as outer decoding when a convolutional code is employed as the outer code. Given the prior probabilities, $P(u_k)$, and channel likelihood estimates, $P[y_k|x_k]$, the log-domain computations of the BCJR algorithm are divided into three groups:

1) Branch metric computation for each branch between states s_{k-1} to s_k:

$$\begin{aligned}\gamma_k(s_{k-1}, s_k) = \ln\{P(u_k = f(s_{k-1}, s_k)\}\\ + \ln\{P[y_k|x_k = g(s_{k-1}, s_k)]\}.\end{aligned} \quad (1)$$

2) Forward/Backward iteration for each state, s_k, assuming a radix-2 trellis: Forward state metric; valid transitions are (s_{k-1}, s_k), (s'_{k-1}, s_k):

$$\alpha_k(s_k) = \ln\left\{\begin{array}{l}\exp[\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)]+\\ \exp[\alpha_{k-1}(s'_{k-1}) + \gamma_k(s'_{k-1}, s_k)]\end{array}\right\}. \quad (2)$$

Backward state metric; valid transitions are (s_k, s_{k+1}), (s_k, s'_{k+1}):

$$\beta_k(s_k) = \ln\left\{\begin{array}{l}\exp[\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})]+\\ \exp[\beta_{k+1}(s'_{k+1}) + \gamma_{k+1}(s_k, s'_{k+1},)]\end{array}\right\}. \quad (3)$$

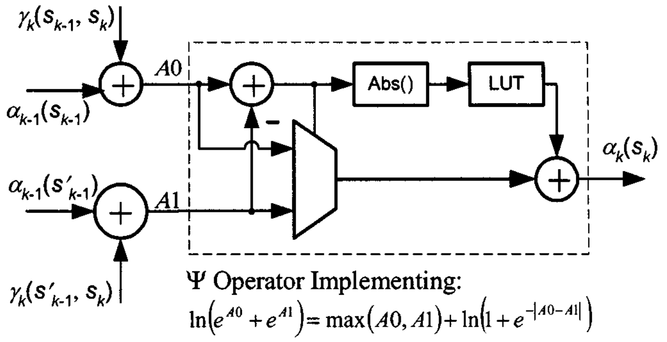

Fig. 5. Add-compare-select unit for an iterator (either forward or backward) using the Ψ (.) operator as indicated within the box.

3) Depending on position (inner/outer) of decoder, the required *a posteriori* probability, $\Lambda(z_k)$ is either $\ln(P[u_k|y])$ or $\ln(P[x_k|y])$ respectively.

$$\Lambda(z_k) = \ln\left\{\sum_{\forall\, s = s_k(z_k)} \exp[\alpha_k(s) + \beta_k(s)]\right\}$$
$$z_k \in \{+1, -1\}. \quad (4)$$

The structures for both forward and backward iterations are identical, and similar to the Add–Compare–Select units used in Viterbi decoders. Thus only the forward iterator (Fig. 5) will be described. The current branch metrics (γ_k) are added to the corresponding state metrics (α_k) from the previous iteration:

$$\begin{aligned}A_0 &= \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)\\ A_1 &= \alpha_{k-1}(s'_{k-1}) + \gamma_k(s'_{k-1}, s_k).\end{aligned} \quad (5)$$

The logarithm of the sum of exponentials is then evaluated with a new operator, Ψ. It uses a comparator, a lookup table, and a final adder (Fig. 5) to approximate the second term in the equation [10]:

$$\alpha_k(s_k) = \ln(e^{A_0} + e^{A_1}) = \max\{A_0, A_1\} + \ln(1 + e^{-|A_0 - A_1|}). \quad (6)$$

The forward/backward iteration structures are therefore termed the Add-Compare-Select-Add (ACSA) units. A number of Ψ operators are also used in the computation of the *a posteriori* values using a tree structure shown in Fig. 8.

To implement the original BCJR algorithm, the backward iteration can only begin after complete observation of the block of 4k bits, resulting in large memory requirements and long latencies. Variations of the BCJR algorithm avoid these effects by windowing or limiting the number of backward iteration steps.

A. Backward Propagation of Windowed BCJR

An implementation of windowed BCJR with asymptotically equivalent performance can be achieved using two overlapping windows for the β-computation. Each window spans a width of $2L$, and overlaps with the other window in both trellis position and time by L steps, as shown in Fig. 6. The initial L outputs are always discarded while the latter L outputs, having satisfied a criterion for minimum number of steps, L, through the trellis, are retained and eventually combined with the appropriate α

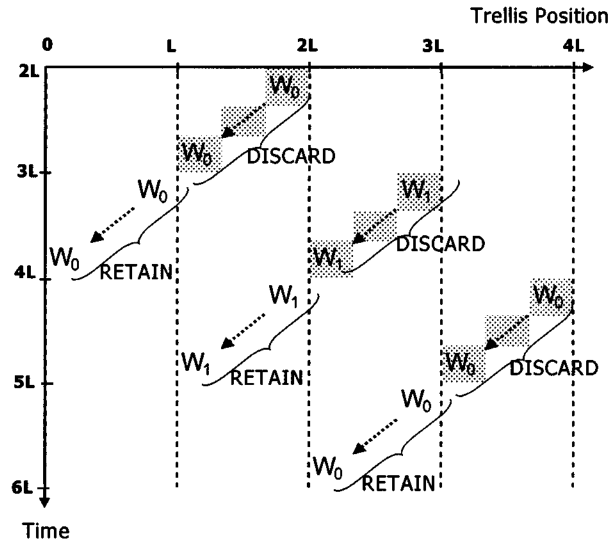

Fig. 6. Backward iteration using 2 overlapping windows, W_0 and W_1 for BCJR algorithm. The shaded outputs are always discarded.

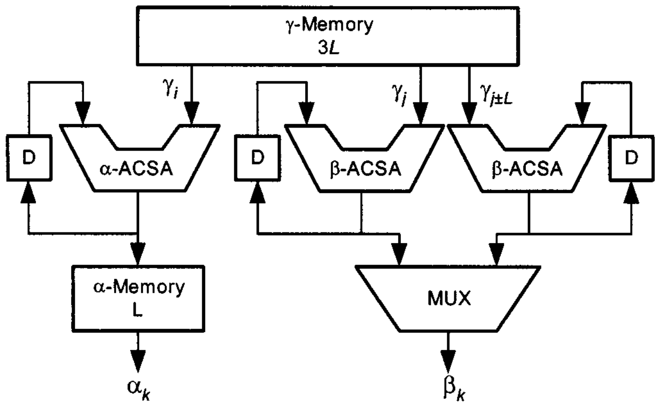

Fig. 7. State-slice of a MAP decoder structure.

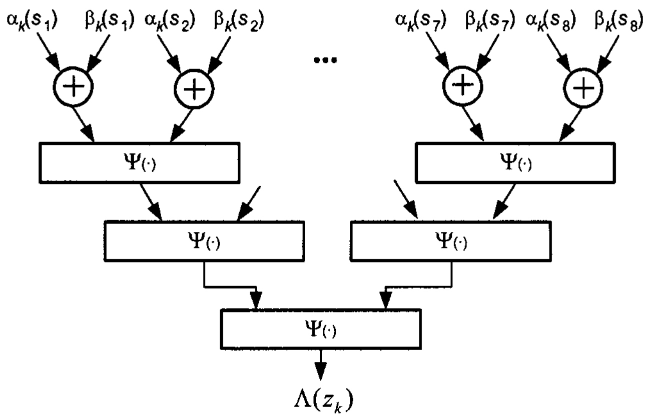

Fig. 8. Λ block makes use of a binary tree of Ψ (.) operators.

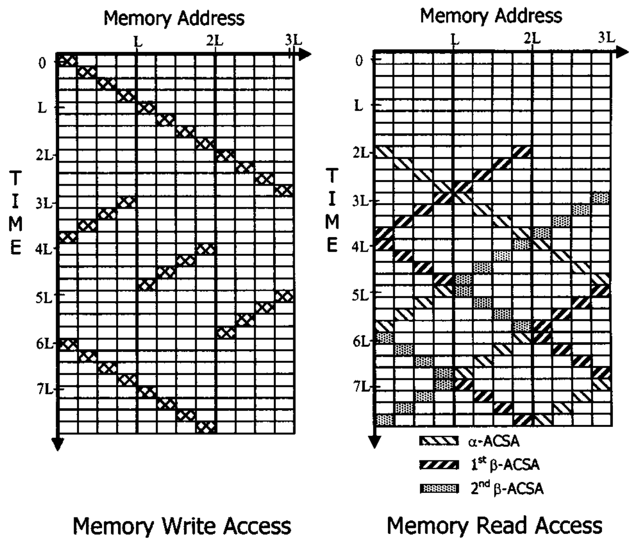

Fig. 9. Memory read and write access of branch metrics γ_k.

values to obtain the soft outputs. This scheme results in lower memory requirement and less computational hardware.

Fig. 7 shows a state-slice of the MAP decoder that is able to maintain a throughput equal to the input arrival rate. The γ-memory stores the branch metrics. An α-ACSA performs the forward iteration and stores its outputs in the α-Memory. Two β-ACSA's perform the backward iteration in accordance with the overlapping window method.

Fig. 9 reproduces the timing diagram of a scheme that would limit the interval between the production and 3 consumption cycles to $3L$. The implementation partitions each γ-memory block into 3 sections of length L (3 sections of L columns in Fig. 9) and deliberately delays the first forward iteration by $2L$. New data is cyclically written into each of the partitions while the write/read access pattern within each partition is continuously alternated between left-to-right and right-to-left directions every L periods. Each branch metric entry, γ_k, in memory is read once by each of the three ACSA's. After the third and final read access, the memory location is immediately replaced with new data. The repetitive nature of the memory access within each partition promotes reduction in control logic, compared to random access memory, and is implemented as a bi-directional shift register.

Similarly, observations on the production and consumption patterns of the α values will indicate that each α-Memory block can be implemented with a bi-directional L-word shift register.

Finally, evaluation of (4), the *a posteriori* result, is performed by summing the α and β values in a tree structure (Fig. 8), and a final adder evaluates the log-likelihood ratios.

Although the maximum latency through each MAP decoder is $4L$ (80 cycles for $L = 20$), it remains insignificant compared with that of the interleaver discussed in Section II.

IV. Soft Output Viterbi Algorithm (SOVA) Decoder

The computational complexity of the BCJR algorithm can be traded for reduced BER performance by replacing the MAP decoders with SOVA decoders [3].

As in BCJR, a windowed SOVA is advantageous in terms of its memory requirement and latency, when compared to the full SOVA. Previous windowed SOVA implementations [12] made use of a two-step algorithm (Fig. 10). The first stage is a regular

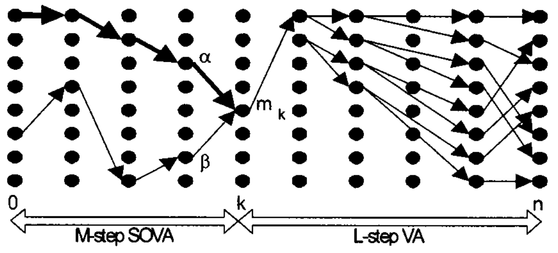

Fig. 10. Realization of a SOVA decoder by cascading a typical VA survival memory unit with a SOVA section. α and β are the two most likely paths that will arrive at state m_k.

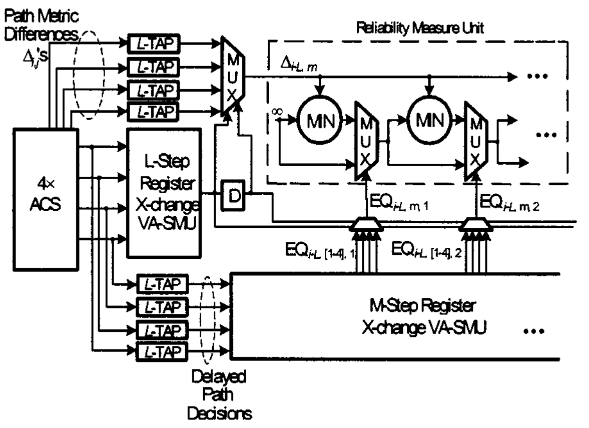

Fig. 11. Example 4-state system block for SOVA.

L-step windowed-Viterbi algorithm (VA) that obtains the most likely state, m_k, with a delay of L. This is followed by another M-step traceback to find the two most likely paths arriving at m_k. Tracebacks are performed by recursively reading intermediate decisions that were stored in an SRAM.

The SRAM-based traceback has a costly implementation complexity due to address decoder and sense amplifier overhead. An implementation of SOVA combining the efficiency of a register exchange pipeline with the two-stage SOVA is presented in Fig. 11.

From each of the ACS's, the difference between the two path metrics, $\Delta_{i,j}$, arriving at time i, state j is retained. Additionally, a modified register exchange (Fig. 12) provides $\mathrm{EQ}_{i-L,j,k}$ outputs indicating the equality between the competing decisions at time $i-L$, state j, from which a traceback of depth k is initiated. Using the decisions from the VA-SMU, $\Delta_{i,m}$'s and $\mathrm{EQ}_{i-L,m,k}$'s corresponding to the most likely state, m, are multiplexed into the Reliability Measure Unit (RMU), which uses comparators (minimizing function) and multiplexers in a pipeline to select the minimum Δ_i along the most-likely path. The pipeline is initialized with the maximum reliability measure allowed by the particular binary representation (conceptually represented as "∞" in Fig. 11). Based on the EQ input, each pipelined section outputs one of the following:

1) Equal decision—reliability measure from the previous step.
2) Different decision—minimum of Δ_i and reliability measure from the previous step.

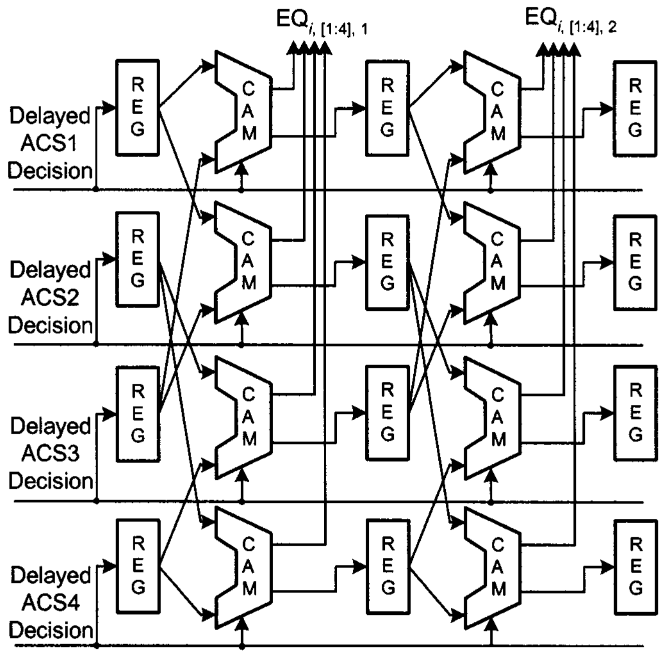

Fig. 12. Example 4-state SOVA-Register Exchange Survival Memory Unit with Compare-and-Mux (CAM) units to perform equivalence checking and multiplexing. The outputs $\mathrm{EQ}_{i,j,k}$ indicate equality of decisions taken at step i, state j, and traceback depth k.

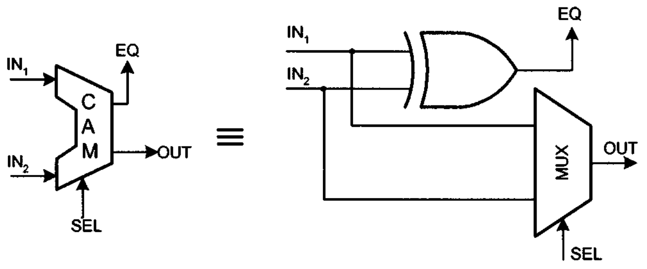

Fig. 13. Block diagram of Compare-and-Mux (CAM) unit comprising an XOR gate for equality checking and multiplexing function.

Compared with a hard-output Viterbi decoder implementation, the total size of the SMU's is approximately doubled (assuming the difference between L and M is small). The RMU overhead consists of M copies of 1 register, 2 multiplexers and a 2-input comparator performing the minimization function.

The latency through the SOVA decoder is $L+M$. This remains insignificant compared with the overall latency in the Turbo-SOVA system, which is dominated by the latency through the interleavers.

V. Low Density Parity Check Code Decoders [4]

An LDPC code with a 512×4608 parity check matrix will be used as an example outer code. This parity check matrix has columns of weight 4 and rows of weight 36, and comprises a total of $4608 \times 4 = 18\,432$ nonzero entries. The parity check information is also commonly represented in a bi-partite graph as 512 check nodes and 4608 bit nodes.

The following log-domain equations modified from [5] exploit the large number of common terms in each group of computations.

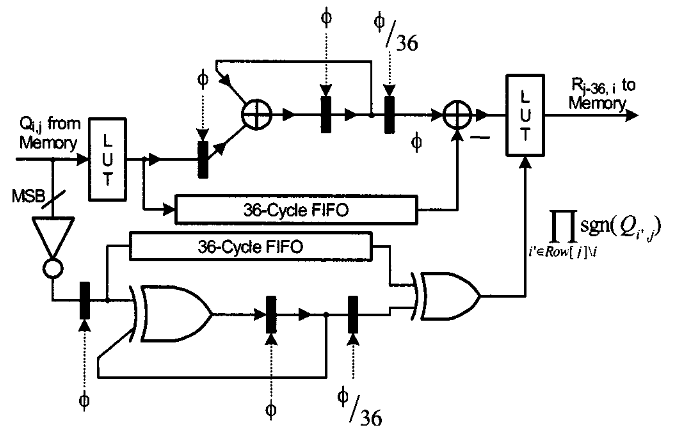

Fig. 14. Recursive pipelined implementation to compute check-to-bit messages.

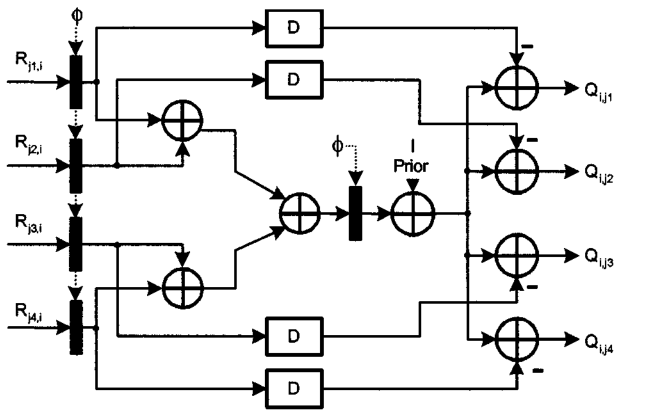

Fig. 15. Binary adder tree to compute bit-to-check message in a 2-stage pipeline.

1) Check j to bit i messaging (parity check):

$$R_{j,i} = \Phi^{-1}\left\{\left(\sum_{i' \in Row[j]} \Phi(Q_{i',j})\right) - \Phi(Q_{i,j})\right\} \times \left(\mathrm{sgn}(Q_{i,j}) \bullet \prod_{i' \in Row[j]} \mathrm{sgn}(Q_{i',j})\right) \bullet (-1)^{|Row[j]|} \tag{7}$$

where Φ and Φ^{-1} are evaluated using lookup-tables:

$$\Phi(x) = -\log(\tanh(1/2x))$$
$$\Phi^{-1}(x) = 2\tanh^{-1}[\exp(-x)]. \tag{8}$$

A simple expansion of these terms will show that $\Phi(x) = \Phi^{-1}(x)$, implying that the implemented lookup-tables are identical.

2) Bit i to check j messaging:

$$Q_{i,j} = \left(\sum_{j' \in Col[i]} R_{j',i}\right) - R_{j,i} + \ln\left[\frac{p_i(1)}{p_i(0)}\right] \tag{9}$$

where $\ln[p_i(1)/p_i(0)]$ is the prior information for bit i.

A. Check to Bit Message Computation

The example LDPC block code has a total of 512 parity checks, where each parity check j computes $R_{j,i}$ using entries from 36 bit nodes $i_1, i_2, i_3, \ldots, i_{36}$. The bottleneck in (7) is the 36-input summation, $\sum_{i \in Row[j]} \Phi(Q_{i,j})$.

In order to maintain a high throughput with a small number of read ports, the computation is performed with P copies of structures identical to Fig. 14. They are cascaded in parallel to achieve a throughput of $P \times R_{j,i}$ messages per cycle. A natural choice for the value of P would be the column weight (4 in our example), such that all check-to-bit messages are computed in approximately the same time it would have taken to acquire a new block of inputs.

Using 2's complement representation, the most-significant bit (MSB) of the messages is a sign bit. Therefore, the product, $\prod_{i' \in Row[j] \setminus i} \mathrm{sgn}(Q_{i',j})$, is equal to a collective XOR of the MSB of all inputs. The result is fed into the output LUT to direct an $R_{j,i}$ output with the appropriate sign. In addition, the final lookup table could be precoded to account for the deterministic term, $(-1)^{|Row[j]|}$, in (7).

B. Bit to Check Message Computation

Each of the 4608 bit nodes in the example LDPC decoder computes $Q_{i,j}$ using entries from 4 different check nodes j_1, j_2, j_3 and j_4. The bottleneck is the 4-input summation: $\sum_{j \in Col[i]} R_{j,i}$.

Unlike the earlier 36-input summation, the small number of inputs makes it very suitable for a pipelined tree adder structure as shown in Fig. 15. With a steady state throughput of four $Q_{i,j}$ messages per cycle, the total latency to compute all the messages is again approximately the same time it would have taken to acquire a new block of inputs.

C. LDPC Memory Design

While the computational complexity of an LDPC decoder is very low compared with the MAP or SOVA decoders, the memory requirement far exceeds those of the latter two. Due to the irregularity in the parity check matrix, the two classes of computations over a single block of inputs, bit-to-check and check-to-bit, cannot be overlapped. In order to achieve fully pipelined throughputs, each memory block in the LDPC decoder is implemented as two buffers alternating between read/write. Thus, for a single iteration of LDPC decoding (bit-to-check and vice-versa) the required memory is $4 \times 18\,432 = 73\,728$ words. This section provides a proposal for structural indexing of the messages to simplify the control logic.

The $R_{j,i}$ messages are indexed in a 2-D array with j indices that are ordered sequentially in the y-direction and strictly increasing i indices in the x-direction. Fig. 17 shows an example 512×36 matrix of $R_{j,i}$ messages. In general, the $R_{j,i}$ messages are not consumed in any particular order along the j indices (or y-direction). With the described arrangement though, entries along each row are consumed in a strictly left-to-right manner. Thus each 36-entry row of the matrix is stored in a first-in-first-out (FIFO) buffer, which also removes the requirement for the $Q_{i,j}$ computation block to keep track of the column index. Inputs are simply indexed by their row numbers, and

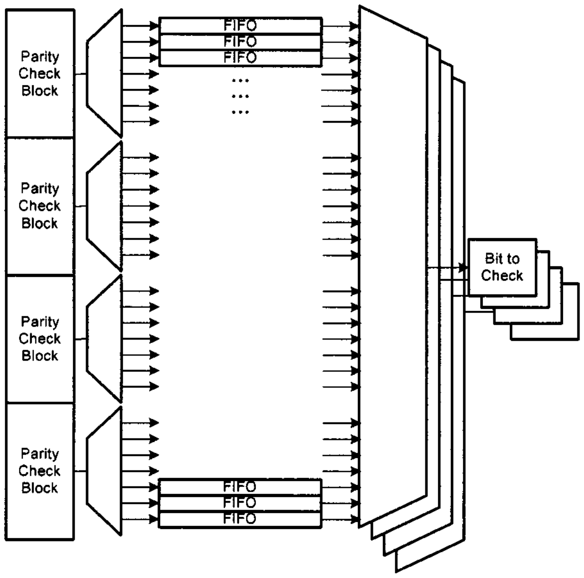

Fig. 16. Using FIFO's to store rows of $R_{j,\,i}$ messages.

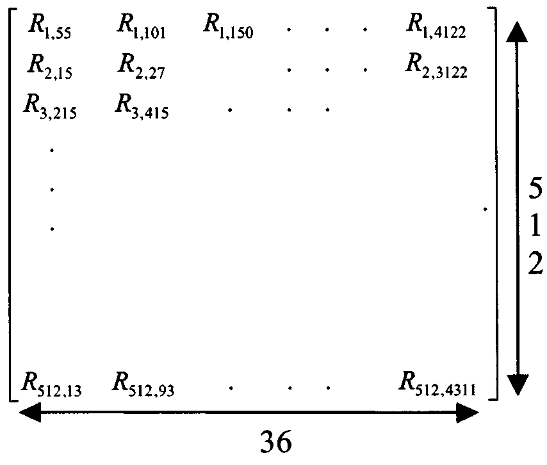
$R_{1,55}$ $R_{1,101}$ $R_{1,150}$ $R_{1,4122}$
$R_{2,15}$ $R_{2,27}$ $R_{2,3122}$
$R_{3,215}$ $R_{3,415}$
$R_{512,13}$ $R_{512,93}$ $R_{512,4311}$
512
36

Fig. 17. Example 512×36 memory array for $R_{j,\,i}$ values.

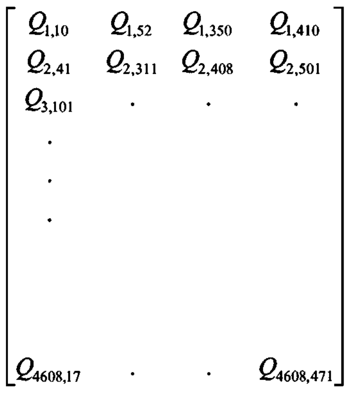
$Q_{1,10}$ $Q_{1,52}$ $Q_{1,350}$ $Q_{1,410}$
$Q_{2,41}$ $Q_{2,311}$ $Q_{2,408}$ $Q_{2,501}$
$Q_{3,101}$
$Q_{4608,17}$ $Q_{4608,471}$

Fig. 18. Example 4608×4 memory array for $Q_{i,\,j}$ values.

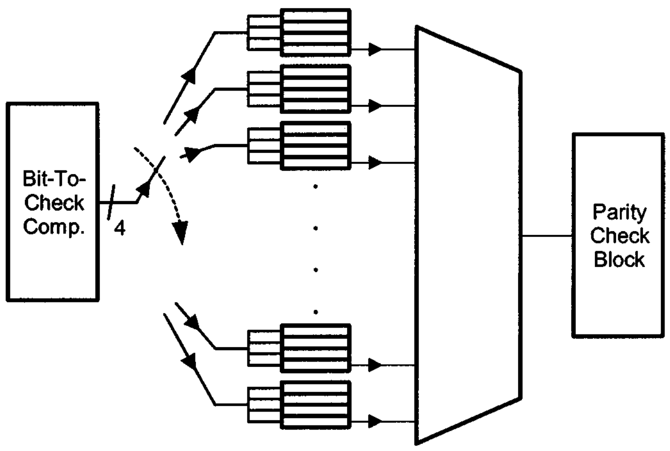

Fig. 19. Using 4-input stacks to store rows of $Q_{i,\,j}$ messages.

each read port can therefore be implemented as a 512-input multiplexer.

Fig. 16 shows that each of the 4 parity check blocks outputs to a quarter of all the FIFO's. The demultiplexer-select is incremented once every 36 cycles to switch to the next FIFO, which stores the next row in the $R_{j,\,i}$ matrix.

Similarly, $Q_{i,\,j}$ is indexed as shown in Fig. 18, with i indices that are ordered sequentially in the y-direction and strictly increasing j indices in the x-direction. Each row of the $Q_{i,\,j}$ matrix is stored in a 4-entry stack. The 4 messages are produced simultaneously by the tree adder structure described previously, but consumed in a strictly left-to-right manner. The subsequent parity check ($R_{j,\,i}$) computations only need to keep track of the row numbers to pop the correct values from the appropriate stack through a 4608-input multiplexer. A pipelined multiplexer is necessary in order to meet Gbps throughtputs with such a large number of inputs.

VI. Comparison of SISO Decoders

The number of computational units required for each of the iterative decoder modules is summarized in Table I. An E^2PR4 channel decoder is concatenated with either a 16-state binary convolutional decoder or an LDPC decoder. As described in Section IV, the BCJR decoder can be replaced with a SOVA decoder. The number of ACSA units is reduced from 3 to 1 per state-slice, a 66% savings in structural computation units, while memory savings is 30%. The SOVA algorithm trades off complexity for predictable degradation in BER performance over the BCJR algorithm.

The throughputs of both the MAP and SOVA decoders are limited by the feedback loop that exists in the Add–Compare–Select units. If area and power were not constrained, a 1 Gbps iterative decoder based on MAP or SOVA decoding would be achievable with current technology; however, due to mandatory unrolling and pipelining, such a decoder will be between 10 and 15 times the area and power of any existing decoder implementation based on conventional Viterbi sequence detection.

On the other hand, the proposed LDPC decoder system is strictly feedforward; therefore, introducing additional levels of pipelining can alleviate delay issues, at the expense of register area and negligible latency. It has been widely recognized that LDPC decoders enjoy significant advantage in terms of computational complexity compared to the trellis-based decoding in MAP and SOVA decoders. This characteristic is reflected in our

TABLE I
COMPUTATIONAL UNITS AND MEMORY REQUIREMENTS FOR ITERATIVE DECODER MODULES

Algorithm	Interleaver	BCJR		SOVA		LDPC
		E^2PR4 Channel Decoder	Convolutional Outer Decoder	E^2PR4 Channel Decoder	Convolutional Outer Decoder	LDPC Outer Decoder
No. of States in Trellis	-	16	16	16	16	-
No. of Output Levels	-	7 [0, ±1, ±2, ±3]	2 [0, 1]	7 [0, ±1, ±2, ±3]	2 [0, 1]	-
No. of Adders *Assumption: comparators are implemented as adders	-	249 γ-Blk 12 α ACSA 64 β ACSA 128 Λ-Blk 45	241 γ-Blk 4 α ACSA 64 β ACSA 128 Λ-Blk 45	60+M γ-Blk 12 ACS 48 RMU M	52+M γ-Blk 4 ACS 48 RMU M	16 $R_{j,i}$ comp. 8 $Q_{i,j}$ comp. 8
No. of LUTs	-	62 α ACSA 16; β ACSA 32; Λ-Blk 14		-	-	8 $R_{j,i}$ comp.
Memories *L: Length of BCJR or VA window *M: Length of SOVA traceback window	2 × 4k words SRAM	$(7L+3) \times 16$ words in shift registers ACSA feedback: 3 × 16 words α-memory: $L \times 16$ words γ-memory: $3L \times 2 \times 16$ words		$[M+(L+1) \times 16]$ words + $[(M + 2L) \times 16]$ bits in shift registers ACS feedback: 16 words VA-SMU: $L \times 16$ bits SOVA-SMU: $M \times 16$bits L-tap Delays: $(L \times 16)$ words + $(L \times 16)$ bits RMU: M words		4 × 18k words in multi-ported FIFO stacks.

proposed implementation, which uses a small number of computational units: 16 adders and 8 LUT's.

However, the lack of any structural regularity in the parity check matrix results in memory requirements that are 2 orders of magnitude larger than those in the MAP or SOVA decoders. It was shown by example in Section V-C that a single LDPC iteration would have a memory requirement upwards of 73 000 words. To make an LDPC decoder implementation more feasible, it will be necessary to introduce regularity into the parity check matrix. Recent publications, [13], [14], suggesting the construction of LDPC-like codes based on difference-set cyclic codes may provide the necessary foundation for building a practical LDPC decoder with reduced memory requirement.

The memory problem is not restricted to LDPC decoders. Interleavers, which are necessary between concatenated convolution decoders, also require significant memory due to the randomness of the output sequences. Interleavers that allow some form of ordered permutation and compact representation will permit efficient implementations of Turbo decoders with no performance loss.

Finally, an iterative decoder implementation for magnetic storage application requires timing recovery methods that can tolerate the increased latencies through multiple decoding iterations.

VII. CONCLUSION

We have proposed datapath-intensive architectures as well as timing and data arrangement schedules for each kind of SISO decoder in order to minimize the critical path delay and simplify the control logic.

Unrolling and pipelining of iterative decoders is necessary to sustain high throughputs, but leads to a linear increase in implementation complexity; however, it provides an excellent opportunity for reduced complexity implementations. Since decisions become increasingly confident after each stage, decoders that are later in the pipeline can trade off some BER performance for reduced complexities. A number of choices are available, ranging from replacing MAP decoders with SOVA decoders or using shorter window lengths to trellis pruning in the trellis-based decoders [15].

The immediate difficulty with LDPC decoders lies in the memory requirement, which should be addressed by designing structured LDPC codes. Without removing the memory bottleneck, further reduced-complexity LDPC decoding, such as approximating the summations in (7) and (9) with minimum and maximum functions respectively, would have little effect on the overall decoder implementation.

REFERENCES

[1] T. Souvignier, M. Oberg, P. Siegel, R. Swanson, and J. Wolf, "Turbo decoding for partial response channels," *IEEE Trans. Commun.*, vol. 48, no. 8, Aug. 2000.

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

[3] J. Hagenauer and L. Papke, "Decoding turbo codes with the soft output viterbi algorithm (SOVA)," in *Proc. IEEE ISIT 1994*, Trondheim, Norway, June 1994, p. 164.

[4] R. G. Gallager, "Low density parity check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[5] J. Fan and J. Cioffi, "Constrained coding techniques for soft iterative decoders," in *Proc. GLOBECOM '99*, vol. 16, Rio de Janeiro, Brazil, Dec. 1999, pp. 723–727.

[6] G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. VLSI Systems*, vol. 7, no. 3, Sept. 1999.

[7] Y. Wu and B. Woerner, "The influence of quantization and fixed point arithmetic upon the BER performance of turbo codes," in *Proc IEEE VTC 1999*, Houston, TX, USA, May 1999, pp. 1683–1687.

[8] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," JPL, TDA Progress Rep., Aug. 1995.

[9] K. Andrews, C. Heegard, and D. Kozen, “Interleaver design methods for turbo codes,” in *Proc. IEEE ISIT 1998*, Cambridge, MA, USA, Aug. 1998, p. 420.

[10] P. Robertson, E. Villebrun, and P. Hoeher, “A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain,” in *Proc. IEEE ICC 1995*, Seattle, WA, USA, Jun. 1995, pp. 1009–1013.

[11] A. Viterbi, “An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes,” *IEEE J. Select. Areas Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[12] C. Berrou, P. Adde, E. Angui, and S. Faudeil, “A low complexity soft-output viterbi decoder architecture,” in *Proc. IEEE ICC 1993*, Geneva, Switzerland, May 1993, pp. 737–740.

[13] D. J. C. Mackay and M. C. Davey, “Evaluation of gallager codes for short block length and high rate applications,” in *Proc. IMA Workshop on Codes, Systems and Graphical Models 1999*, Minneapolis, MN, USA, Aug. 1999.

[14] Y. Kou, S. Lin, and M. P. C. Fossorier, “Low density parity check codes based on finite geometries: A rediscovery,” in *Proc. IEEE ISIT 2000*, Sorrento, Italy, Jun. 2000.

[15] B. Frey and F. Kschischang, “Early detection and trellis splicing: Reduced-complexity iterative decoding,” *IEEE J. Select. Areas Commun.*, vol. 16, no. 2, pp. 153–159, Feb. 1999.