Optimizations of a Turbo-Like Decoder for Deep-Space Optical Communications

M. K. Cheng,[1] M. A. Nakashima,[1] B. E. Moision,[1] and J. Hamkins[1]

The National Aeronautics and Space Administration has developed a capacity-approaching modulation and coding scheme that comprises a serial concatenation of an inner accumulate pulse-position modulation and an outer convolutional code for deep-space optical communications. Decoding of this serially concatenated pulse-position modulation (SCPPM) code uses a turbo-like algorithm. However, the inner code trellis contains many parallel edges that are not typical in standard turbo codes and, therefore, a straightforward application of classical turbo decoding is very inefficient. Here, we present various optimizations applicable in hardware implementation of the SCPPM decoder. More specifically, we feature a Super Gamma computation to efficiently handle parallel trellis edges, a "maxstar top 2" circuit fit for pipelining, a modified two's complement subtraction circuit with a shorter path delay, and a cyclic redundancy check circuit for window-based turbo decoders. We also present a polynomial interleaver where current interleaver positions can be calculated from previous positions. This recursive interleaver property enables an algorithmic realization in which no memory is needed to store the interleaver mappings.

Using the featured optimizations, we implement a 6.72 megabits-per-second (Mbps) SCPPM decoder on a single field-programmable gate array (FPGA). Compared to the current data rate of 256 kbps from Mars, the SCPPM-coded scheme represents a throughput increase of more than twenty-six fold. Extension to a 50-Mbps decoder on a board with multiple FPGAs follows naturally. We show through hardware simulations that the SCPPM-coded system can operate within 1 dB of the Shannon capacity at nominal operating conditions.

I. Introduction

All of the current deep-space missions of the National Aeronautics and Space Administration (NASA) communicate to Earth using the radio frequency (RF) spectrum. However, the RF spectrum contains much congestion and is susceptible to high diffraction loss due to the spreading of the beamwidths. For example, if we use a transmit antenna that is 3.7 m in diameter (such as one that is mounted on Voyager) and a frequency in X-band (approximately 8 GHz) to communicate between Earth and Saturn,

[1] Communications Architectures and Research Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

the transmission beam will spread out over an area over 1000 Earth diameters wide due to diffraction. We can contrast this result with a system that employs optical communications. If we instead use a small 10-cm optical telescope with a wavelength of 1 μm to communicate data between the same Earth–Saturn distance, the resulting spot size will be only 1 Earth diameter wide. This represents a factor of 1000 concentration of received energy in both the horizontal and vertical directions (a factor of 10^6 in power intensity). This improved energy delivery efficiency allows an optical link to operate at a lower transmit power and aperture size while still achieving a higher link data rate. For all of these reasons and more, NASA plans to utilize higher frequency regions in the electromagnetic spectrum to increase the deep-space information throughput from 256 kilobits per second (kbps) (Mars Exploration Rovers) to tens of megabits per second (Mbps) and beyond.

Modulation and error-correction coding are keys to reliable communications. In the case of an optical link with direct detection, which we consider here, a modulation that has a high peak-to-average-power ratio has been shown to be very efficient [1]. Pulse-position modulation (PPM) is one scheme that offers a high peak-to-average-power ratio. An M-order PPM divides a symbol interval into M possible pulse locations and a signal pulse is placed into only one of these possible positions, depending on the information to be transmitted.

Moision and Hamkins compared various concatenated modulation coding schemes with PPM that included Reed–Solomon PPM (RS-PPM), low-density parity-check PPM (LDPC-PPM), and convolutionally coded PPM. They discovered that a serially concatenated pulse-position modulation (SCPPM) scheme offers the best performance and complexity trade-off [2].

Modulation is a mapping of bits to symbols transmitted on the channel. This mapping may be considered a code, and demodulation may be considered decoding of the code. Conventionally, the modulation and error-correcting code (ECC) are decoded independently, with the demodulator sending its results to the ECC decoder. However, the combination of the modulation and the ECC can be treated as a single large code, which maps user information bits directly to the symbols transmitted on the channel. In some instances, several decibels in performance could be gained by decoding the ECC and modulation jointly as a single code as opposed to decoding them independently. An exact maximum-likelihood (ML) decoding of the joint modulation–ECC code would be prohibitively complex in most cases of practical interest. However, true ML decoding may be approximated while limiting the decoder complexity by iteratively decoding the modulation and the ECC. This is in fact the "turbo" principle, and more details can be found in [3].

Due to the unique structure of SCPPM, a straightforward application of the standard turbo decoding algorithm would be very inefficient. Other codes, especially ones designed for the optical channel, that have similar constructions might face the same challenges in their decoding complexity and could benefit from optimizations presented in this work.

This article is organized as follows: in Section II, we give our channel assumptions. In Section III, we describe the SCPPM code construction and explain why application of classical turbo decoding is not practical. The SCPPM decoding algorithm uses the turbo principle but includes many new techniques that optimize the decoding speed and performance. In Section IV, we describe the turbo-like part of the SCPPM decoding. In Section V, we present the hardware optimizations. More specifically, we illustrate how to efficiently decode on a trellis that contains parallel edges. We also describe a hybrid "maxstar top 2" circuit fit for pipelining and a modified two's complement subtraction circuit with a short path delay. In Section VI, we describe the SCPPM interleaver design. The interleaver, characterized by a permutation polynomial, produces a good decoder threshold and a low decoder error floor. The interleaver also has an algorithmic realization that does not require storing the interleaving and deinterleaving mappings.

To increase the overall throughput, the SCPPM code trellis can be partitioned into windows, and parallel decoders can be applied to the windows. In Section VII, we provide a cyclic redundancy check

(CRC) circuit that will work with window-based turbo decoders. In Section VIII, we present various field-programmable gate array (FPGA) implementations of the SCPPM decoder that include the featured optimizations and show that SCPPM can operate within 1 dB of capacity in a nominal deep-space mission scenario. We demonstrate that a 6.72-Mbps decoder can be realized on a single FPGA. In addition, we outline a readily achievable path to implementing a SCPPM decoder that can deliver 50 Mbps (enough to transfer compressed high-definition television signals) and beyond for deep-space or satellite communications.

II. Channel Assumptions

We consider an optical communications system that uses direct photon detection with a high-order PPM [4, Chapter 1.2]. An M-order PPM uses a time interval that is divided into M possible pulse locations, but only a single pulse is placed into one of the possible positions. The position of the pulse is determined by the information to be transmitted. A diagram of the optical communications system under discussion is shown in Fig. 1. The information bits $\mathbf{u} = (u_1, u_2, \cdots, u_K)$ are independent, identically distributed (i.i.d.) binary random variables assumed to take on the values 0 and 1 with equal probability. The vector $\mathbf{u}$ is encoded to $\mathbf{c} = (c_1, c_2, \cdots, c_n)$, a vector of n PPM symbols. The overall length in bits for a codeword block is $N = n \log_2 M$.

At the receiver, light is focused on a detector that responds to individual photons, as illustrated in Fig. 2. For each photon sensed, the detector produces a band-limited waveform for input to the demodulator. This waveform is used to estimate the photon count k_i within each slot i. On the Poisson channel, a nonsignaling slot has average photon count n_b and a signaling slot has average count $n_s + n_b$, so that the likelihood ratio of slot i is calculated by

$$LR(k_i) = \frac{(n_s + n_b)^{k_i} e^{-(n_s+n_b)}/k_i!}{n_b^{k_i} e^{-n_b}/k_i!}$$

$$= e^{-n_s} \left(1 + \frac{n_s}{n_b}\right)^{k_i} \tag{1}$$

More on the receiver design can be found in [5].

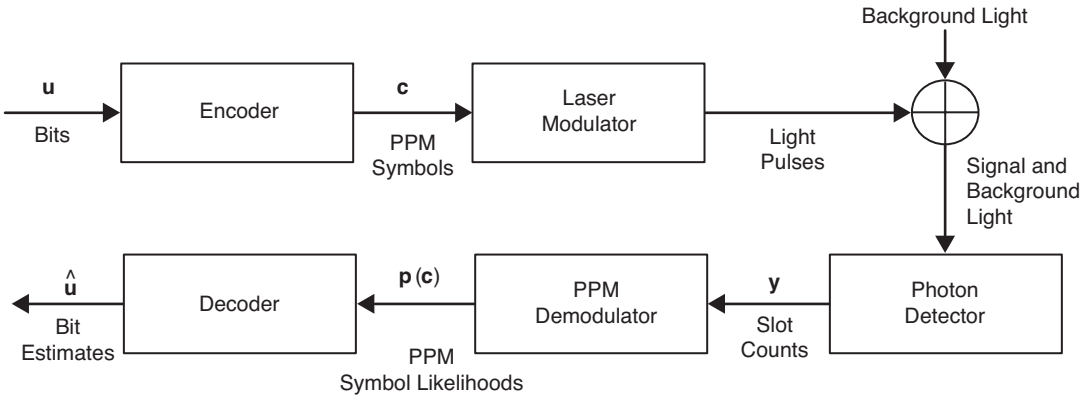

Fig. 1. An optical communications system.

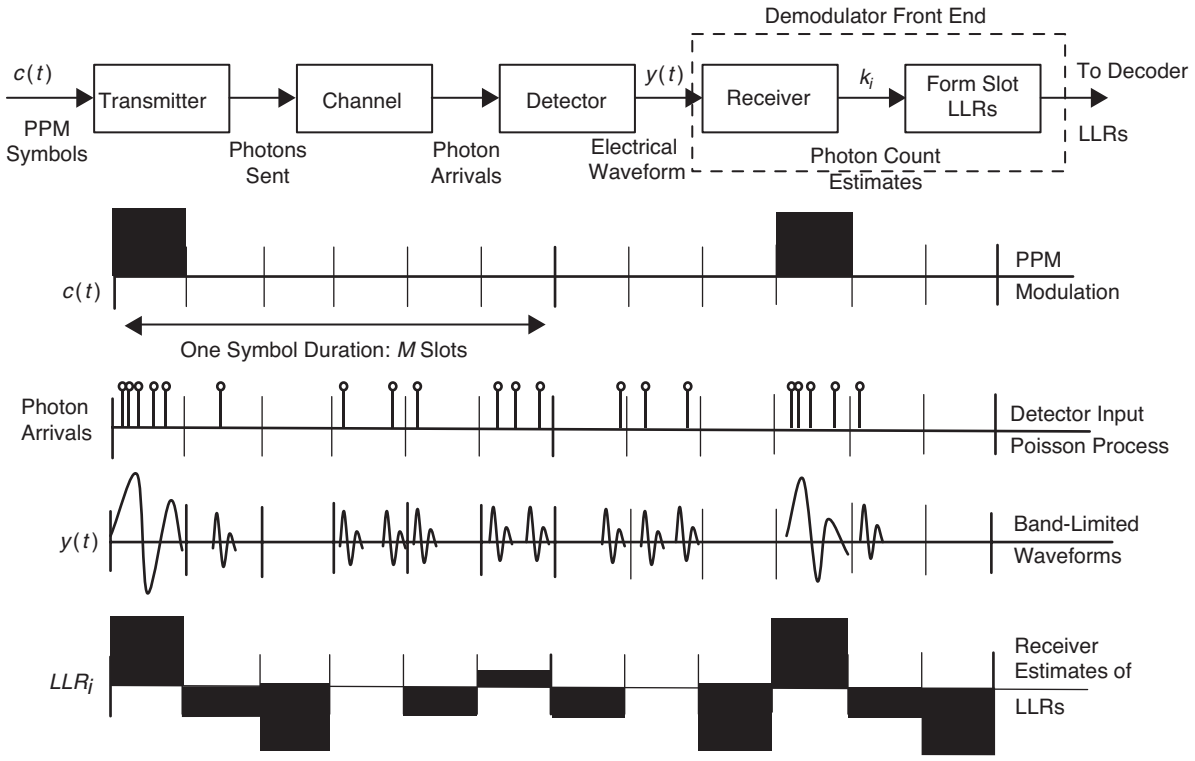

Fig. 2. From PPM symbols to log-likelihood ratios (LLRs).

III. The Serially Concatenated Pulse-Position Modulation Code

The SCPPM encoder, shown in Fig. 3, consists of an outer rate-1/2 constraint-length-3 convolutional code, a permutation polynomial interleaver, and an inner accumulate PPM (APPM) code. A block of K information bits $\mathbf{u}$ that includes a CRC is encoded by the outer convolutional code to yield a length-N coded sequence $\mathbf{x}$. This coded sequence is permuted bit-wise to produce the sequence $\mathbf{a}$ that is then filtered by an accumulator and mapped to $n = N/\log_2 M$ PPM symbols $\mathbf{c}$. There are $\log_2 M$ bits per PPM symbol. Due to the APPM bits-to-symbol mapping, the trellis that describes the inner code consists of 2 states and $M/2$ parallel branches between connecting states. We cannot directly apply standard turbo decoding and treat each of the parallel edges separately because doing so would make pipelining difficult and increase decoding latency.

The interleaver and deinterleaver are described by second-order permutation polynomials, and efficient designs are given in Section VI.

IV. SCPPM Decoding: The Conventional Turbo-Like Part

Decoding of the SCPPM code uses the turbo principle. The decoding procedure also incorporates new techniques and components that are not found in the standard turbo approach in order to optimize hardware implementation. We first discuss the turbo portion of SCPPM decoding and then present the new techniques and components that enabled a high-data-rate FPGA decoder.

A high-level block diagram of the SCPPM decoder is given in Fig. 4. The symbol I indicates input to the constituent decoders, and O indicates output. The inner decoder operates on the modulation

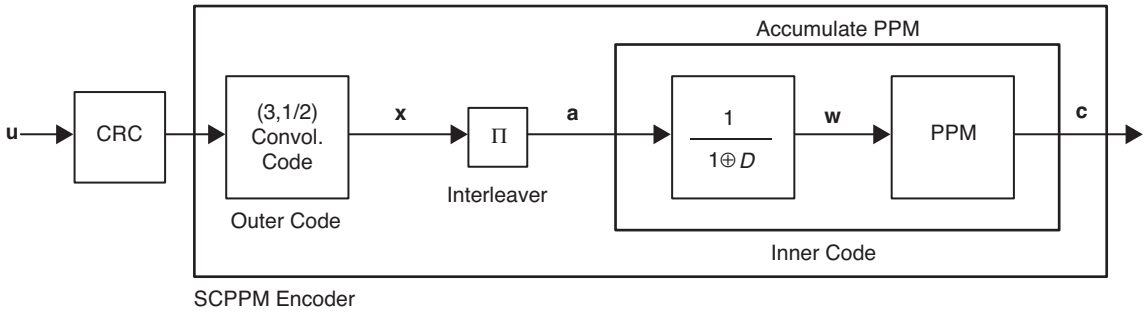

Fig. 3. The SCPPM encoder.

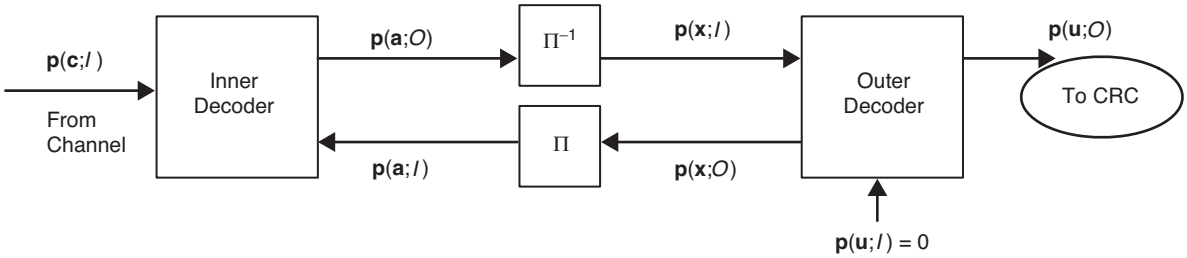

Fig. 4. The SCPPM decoder. Output bits can be directed to a cyclic redundancy check (CRC) to validate codewords.

code, and the outer decoder operates on the convolutional code. Each code is described by a trellis. For each trellis, the Bahl–Cocke–Jelinek–Raviv (BCJR) algorithm [6] is used to compute the a posteriori log-likelihood ratios (LLRs) from a priori LLRs by traversing the trellis in forward and backward directions. Extrinsic information (the difference between the a posteriori and a priori LLRs) is exchanged in iteration rather than the a posteriori LLRs to reduce undesired feedback.

A. Log-Domain Decoding

Each decoder module in the SCPPM decoder applies the BCJR algorithm to the trellis of the constituent code. We use standard notations in the turbo decoding literature [7] and simply restate the calculation of the branch and state metrics inside the inner decoder module. To facilitate hardware realization, the metric computations are done in the log domain [8], which translates multiplications into additions and is less sensitive to round-off errors in fixed-point arithmetic.

Let $\mathcal{V}$ be the set of states and $\mathcal{E}$ be the set of directed labeled edges in a trellis. Each edge $e \in \mathcal{E}$ has an initial state $i(e)$ and a terminal state $t(e)$ (see Fig. 5). For each edge e and stage k of the inner code trellis, the BCJR algorithm traverses the trellis in the backward direction to calculate the log branch metric as

$$\bar{\gamma}_k(e) = p_k(\mathbf{a}; I) + p_k(\mathbf{c}; I) \tag{2}$$

The term $p_k(\mathbf{c}; I)$ is the PPM symbol LLR provided by the channel given in Eq. (1), and the term $p_k(\mathbf{a}; I)$ is the a priori symbol LLR provided by the outer decoder. In the same trellis pass, the BCJR algorithm calculates a backward state log metric for each state s and stage k as

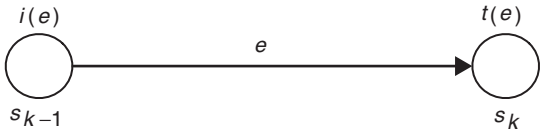

Fig. 5. One stage of a trellis

$$\bar{\beta}_k(s) = \ln \sum_{e:i(e)=s\in\mathcal{V}} \exp\left(\bar{\beta}_{k+1}\big(t(e)\big) + \bar{\gamma}_{k+1}(e)\right) \tag{3}$$

This approach is also known as log maximum a posteriori (log-MAP) decoding [9]. The algorithm then traverses the trellis in the forward direction to calculate the $\bar{\alpha}$'s in the same way. The output LLRs are a function of $\bar{\alpha}$'s, $\bar{\beta}$'s, and $\bar{\gamma}$'s. The outer decoder operates on the trellis that describes the outer code using the same principle.

The log sum of exponentials of Eq. (3) can be expressed as the max of the exponents plus an adjustment term. This operation is known as the maxstar function:

$$\overset{*}{\max}(x, y) \triangleq \ln(e^x + e^y) = \max(x, y) + \ln\left(1 + e^{-|x-y|}\right) \tag{4}$$

The adjustment term can be precomputed and stored in a lookup table to reduce complexity at an increase in memory usage [10]. We can also ignore the adjustment term entirely to save on memory—this approach is known as max log-MAP decoding. Some of the loss incurred from this approximation can be recovered by scaling the extrinsic information that is passed between the inner and outer decoders [11,12].

V. SCPPM Decoding: Optimizations Beyond That of Turbo Conventions

Direct application of conventional turbo decoding to SCPPM is inefficient. Here, we provide new techniques that generate speed, memory, and throughput optimizations. These techniques can be applied to codes that are designed for the optical channel and have structures similar to that of SCPPM.

A. Simplifying Computations with Parallel Trellis Edges

The inner APPM code trellis has 2 states and $2M$ edges per stage and can be viewed as collapsing $\log_2 M$ binary trellis stages into one, as seen in Fig. 6. The forward and backward recursions on this trellis require taking the $\overset{*}{\max}$ of $M/2$ edges per transition between two states. Suppose each 2-input $\overset{*}{\max}$ operation incurs a delay of one clock cycle. A direct implementation of the forward–backward algorithm would require a delay of $\log_2(M/2)$ cycles per transition between two states just for the $\overset{*}{\max}$'s. Barsoum and Moision [2,13] showed that the computation can be pipelined, reducing the $M/2$-input $\overset{*}{\max}$ operation to a 2-input $\overset{*}{\max}$ operation that is computed in one clock cycle. Here, we illustrate how to deal with parallel trellis edges efficiently using the $\bar{\beta}$ recursion. The $\bar{\alpha}$ computation follows in the same manner.

In the product domain, it is straightforward to see an application of the distributive law (multiplication distribution over addition) saves computations on a trellis with parallel edges:

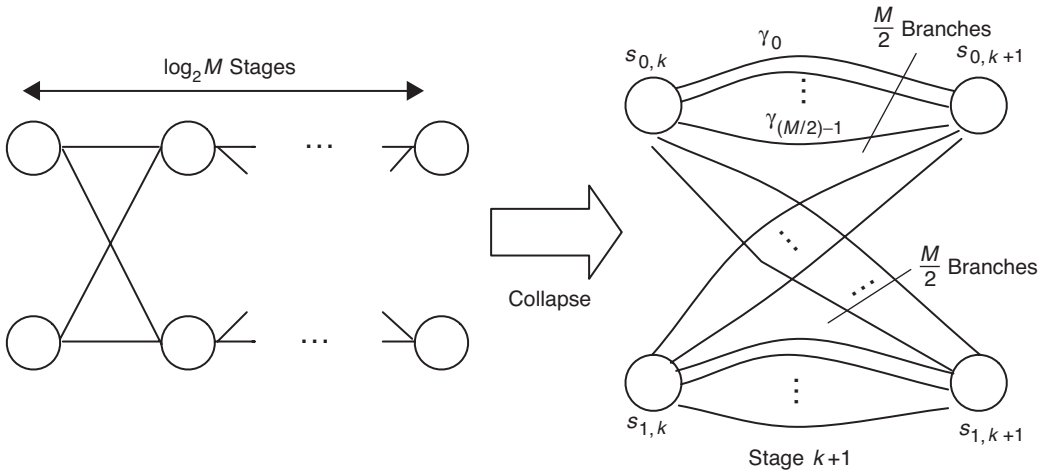

Fig. 6. Grouping $\log_2 M$ bits into one PPM symbol. A single stage of the inner APPM code trellis with $M/2$ parallel edges between connecting states.

$$\begin{aligned}
\beta_k(s) &= \sum_{e:i(e)=s} \beta_{k+1}\big(t(e)\big)\gamma_{k+1}(e) \\
&= \sum_{e:i(e)=s,t(e)=s} \beta_{k+1}(s)\gamma_{k+1}(e) + \sum_{e:i(e)=s,t(e)=\bar{s}} \beta_{k+1}(\bar{s})\gamma_{k+1}(e) \\
&= \left(\beta_{k+1}(s) \sum_{e:i(e)=s,t(e)=s} \gamma_{k+1}(e)\right) + \left(\beta_{k+1}(\bar{s}) \sum_{e:i(e)=s,t(e)=\bar{s}} \gamma_{k+1}(e)\right) \\
&= \beta_{k+1}(s)\gamma'_{k+1}(s,s) + \beta_{k+1}(\bar{s})\gamma'_{k+1}(s,\bar{s}) \qquad (5)
\end{aligned}$$

where $\gamma'_{k+1}(s,s)$, a sum over parallel edges, is referred to as the Super Gamma for the state pair (s,s) at stage $k+1$; the same calculation can be made for the state pair $(s,\bar{s})$.

We have an analogous simplification in the log domain via the distributive law (addition distributive over $\overset{*}{\max}$), which can be seen by taking logarithms of both sides of Eq. (5):

$$\begin{aligned}
\bar{\beta}_k(s) &= \ln\left(\exp\left(\bar{\beta}_{k+1}(s) + \bar{\gamma}'_{k+1}(s,s)\right) + \exp\left(\bar{\beta}_{k+1}(\bar{s}) + \bar{\gamma}'_{k+1}(s,\bar{s})\right)\right) \\
&= \overset{*}{\max}_{\tilde{s}\in\{s,\bar{s}\}} \left\{\bar{\beta}_{k+1}(\tilde{s}) + \bar{\gamma}'_{k+1}(s,\tilde{s})\right\} \qquad (6)
\end{aligned}$$

where

$$\bar{\gamma}'_k(s,\tilde{s}) = \overset{*}{\max}_{e:i(e)=s,t(e)=\tilde{s}} \left\{\bar{\gamma}_k(e)\right\} \qquad (7)$$

Since the $\bar{\gamma}'_k$'s are not a function of a recursively computed quantity, they may be precomputed via a pipeline, as illustrated in Fig. 7. The schedule for the Super γ calculations is shown in Fig. 8. The pipeline is filled with the first $\log_2 M$ stages of γ's. The decoding BCJR algorithm then starts after the pipeline is filled, and thereafter a set of Super γ values per trellis stage is generated per clock.

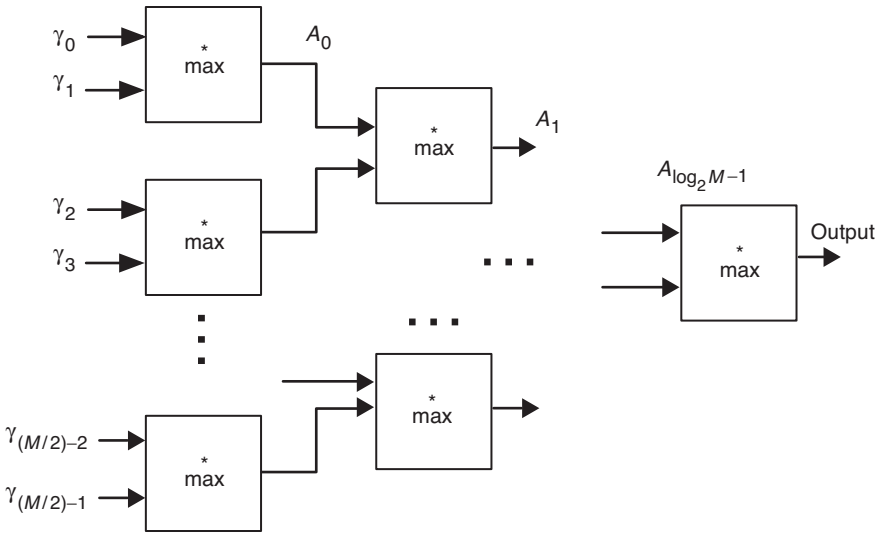

Fig. 7. Pipelined max* to compute the Super γ's.

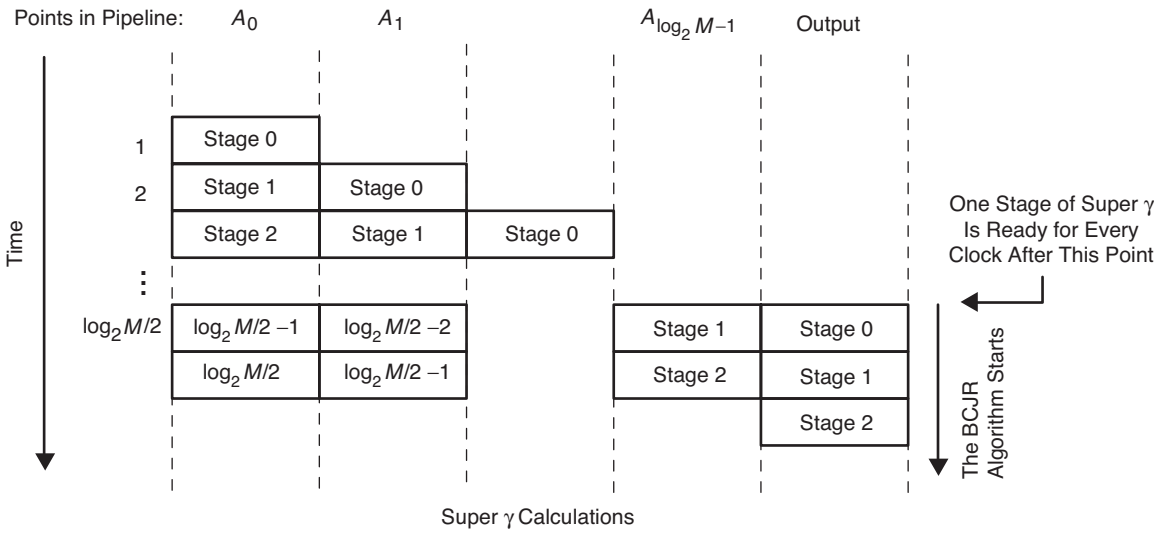

Fig. 8. Scheduling for the Super γ calculations. A stage in the figure represents a stage of γ's in the trellis.

B. Maxstar Only the Top Two Elements

Implementing the full log-MAP decoder consumes much of the FPGA resources because each maxstar operation requires a lookup table. For SCPPM, the number of tables required is increased by the potentially high number of parallel edges in the APPM code trellis. To reduce the FPGA resource utilization, we can ignore the adjustment term in the maxstar function and simply use the max operation. However, this simplification, called max log-MAP decoding, comes with a significant decoder performance loss. Wu and Pisuk [11] showed that a confidence factor, which we denote as FF ($0 < \mathrm{FF} < 1$), can be used to weight the extrinsic LLRs that are passed between two iterative decoders to recover some of the loss incurred from the max log-MAP approach. To recover yet more of the loss, we consider a hybrid approach that takes the maxstar of the top two elements in the input array to further reduce the gap between log-MAP and max log-MAP decoding.

1. The Algorithm. Given an array of n elements $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, one method of finding the maxstar of the top two elements consists of sorting the array. To do this, one can simply assign two variables, *top* and *top*2, to x_1 and x_2, and then compare the rest of the elements in $\mathbf{x}$ first with *top* and then with *top*2. If the compared element is greater than *top*, we replace *top*2 with *top* and *top* with the element. If the compared element is greater than *top*2 only, we replace *top*2 with the element. This procedure is of complexity $O(n)$; at its completion, the two variables will contain the top two elements of $\mathbf{x}$, and we perform a $\overset{*}{\max}(top, top2)$ to obtain the desired result. This method would not be efficient to realize in hardware as it requires a state machine, takes n clocks for each array, and cannot be pipelined. Thus, we develop a "maxstar top 2" algorithm that can be implemented recursively and does not require significant additional circuitry relative to simply taking the max.

The idea is to build from the "max only" pipeline that finds the top element in $\mathbf{x}$. The base case reduces to taking the max of two elements, x_i and x_j, where both $i, j \in \{1, \cdots, n\}$. Instead of propagating only the max of the two elements after each comparison, we also feed forward their difference $\Delta_{i,j} = |x_i - x_j|$. In this way, at every stage of the pipeline, we would then be able to maintain not only the current maximum element but also its difference with the next largest element compared so far in the pipe.

Our maxstar top 2 algorithm takes in four inputs and produces two outputs. The inputs are two elements to be compared, x_i and x_j, and the difference between each and their next largest element in the previous stage, $\Delta_{i,i'}$ and $\Delta_{j,j'}$.

2. The Circuit. The circuit for the two-element maxstar top 2 is given in Fig. 9. The two inputs are denoted here as A and B. Without loss of generality, assume $A > B$. We need to consider only two cases to see how the circuit works. The output of the top multiplexer (mux) will be A, and the lower mux will select δ_A.

Case 1: $\delta_A > |A - B|$**.** We state that δ_A is the difference between A and its previous comparison, denoted here as A'. Substituting for δ_A, we have $|A - A'| > |A - B|$, and we can strip the absolute values because A is the largest of the three so $A - A' > A - B$ and $A' < B$. The element B is closer to A, and we output $|A - B|$ in the final mux.

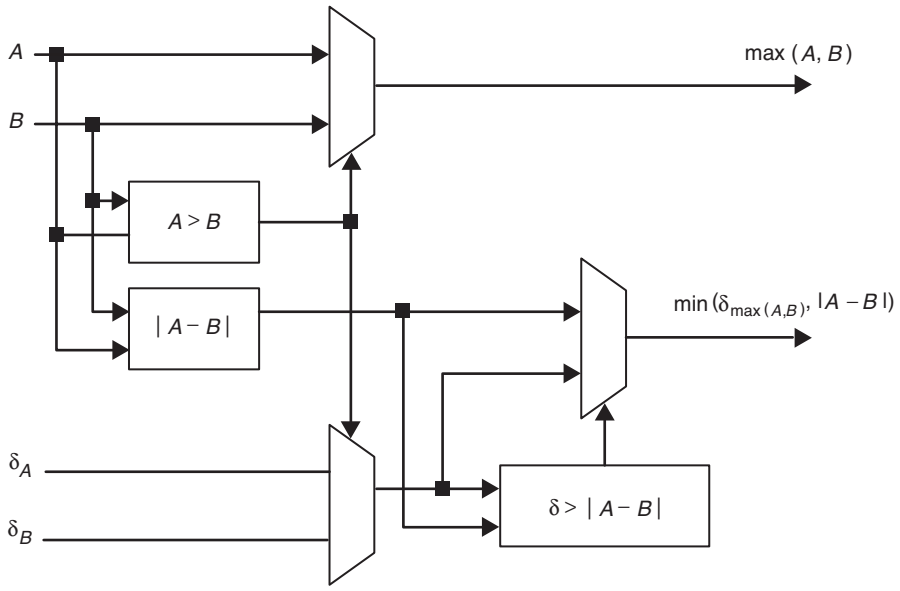

Fig. 9. The maxstar top 2 circuit.

Case 2: $\delta_A \leq |A - B|$**.** We have here $|A - A'| \leq |A - B|$. Stripping the absolute values and rearranging the terms, we get $A' \geq B$. The element A' is closer to A than B, and we output δ_A in the final mux.

We can inductively see that, in the base case with a two-element input, the maxstar top 2 circuit outputs the maximum element and the difference between the maximum and the next largest element in previous comparisons. We can simply replace the maxstar circuit in the pipeline of Fig. 7 with maxstar top 2 to find the largest element and the difference between the largest and the second largest in all stages before the last. We use a maxstar circuit in the last stage to calculate the maxstar of the top two elements in an array, as illustrated in Fig. 10. This approach can be extended to the maxstar of the top four elements and so on. The performance of maxstar top 2 is benchmarked in Section VIII.

C. Fast Modulo Normalization

The BCJR algorithm consists of traversing the code trellis and updating a set of state and branch metrics. The metric update path is illustrated in Fig. 11. Due to the recursive nature of the updates, each of the state metrics in a stage is normalized and clipped by subtracting out the maximum state metric of that stage. This update path cannot be pipelined due to the recursions and becomes the critical path that limits the maximum clock rate at which our design can run on the FPGA. Without normalization, the state metrics can grow unbounded and eventually overflow in a fixed-point hardware implementation. It has been shown for the Viterbi algorithm [10,14] that, as long as the quantization bit width is sufficient to account for the maximum differences between the state metrics, the metrics updates can be allowed to overflow without affecting the result of the computations. This approach naturally extends to the BCJR algorithm, but none of the related literature we found explained this topic in a way that can be easily translated into hardware implementation. Here, we clearly describe this modulo arithmetic approach and provide a block diagram to illustrate its use in SCPPM decoding. We also introduce a modification that can reduce the path delay in this fast modulo operation by 1 bit.

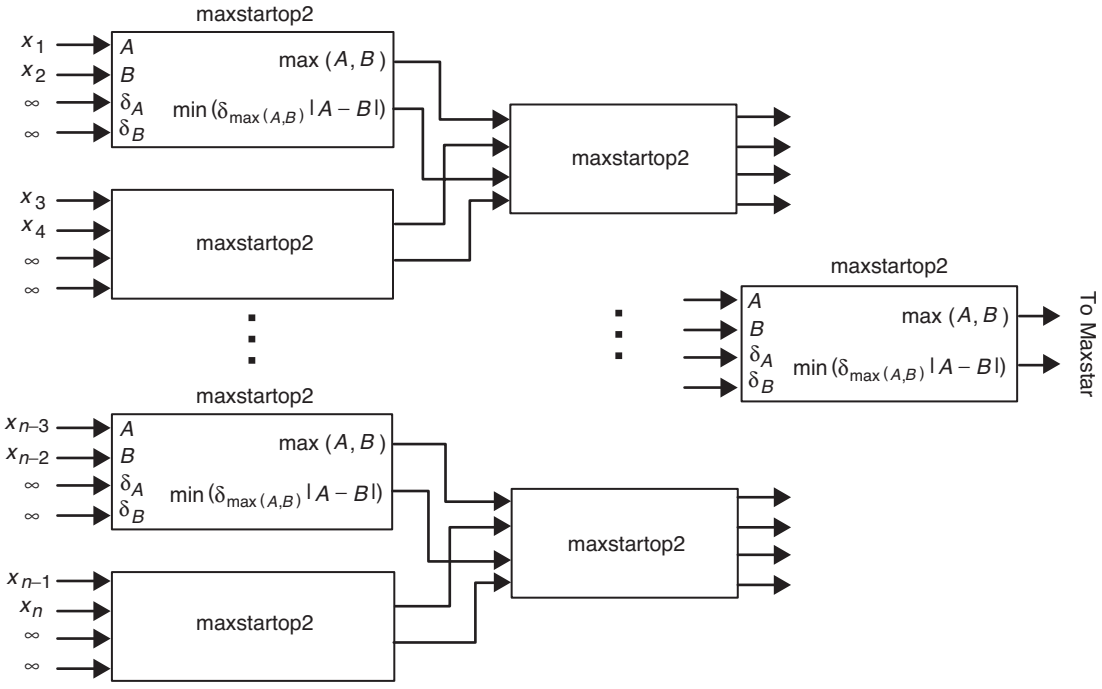

Fig. 10. The maxstar top 2 pipeline.

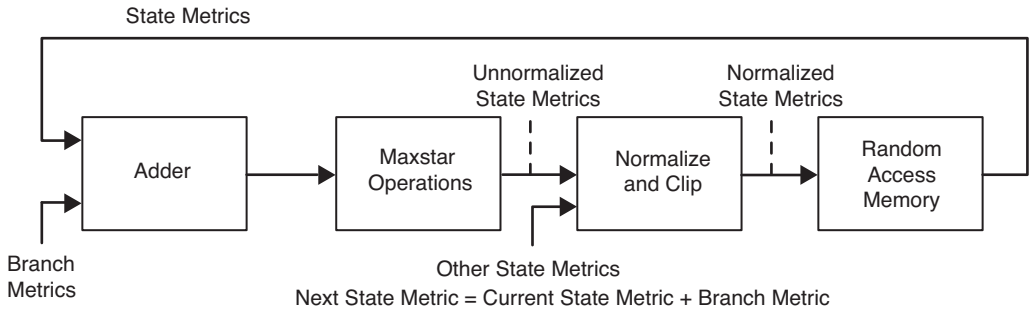

Fig. 11. Metric update in the BCJR algorithm using clipping and normalization. This approach leads to a long critical path.

In modular arithmetic, a metric m_j is mapped into its modulo metric,

$$\bar{m}_j \equiv \left(\left(m_j + \frac{C}{2} \right) \mathrm{mod} C \right) - \frac{C}{2} \tag{8}$$

so that $-(C/2) \leq \bar{m}_j < (C/2)$. This can be visualized by wrapping the real number line around a circle with circumference C. Going around the circle in a counter clockwise direction traverses a path in increasing magnitude, and going around the circle in a clockwise direction traverses a path in decreasing magnitude.

For any two real numbers m_i, m_j such that their absolute difference is bounded by some finite value, that is

$$|m_i - m_j| < \frac{C}{2} \tag{9}$$

their modular difference $|\overline{\bar{m}_i - \bar{m}_j}|$ equals their actual difference $|m_i - m_j|$. Proofs are given in [14,15]. A general description of modulo metric normalization is given in Fig. 12. The angle α is the result of two's complement subtraction of $\bar{m}_2$ from $\bar{m}_1$. We know that $m_1 < m_2$ because $\alpha < \pi$ and its sign bit is 1. The angle α' is the result of two's complement subtraction of $\bar{m}_1$ and $\bar{m}_3$. We know that $m_1 > m_3$ because $\alpha' \geq \pi$ and its sign bit is 0.

To reduce the critical path, we can replace the metric update operation illustrated in Fig. 11 with that of Fig. 13, which uses modulo metric normalization. We see that the new data path avoids the normalization and clipping circuit and consists of only two's complement additions and subtraction.

D. A Fast Two's Complement Subtraction

Shung et al. [15] developed a modified rule that removes a 1-bit delay in two's complement comparison with the Viterbi algorithm in mind. We use their technique as a base for developing a fast two's complement subtraction. Shung defined the comparison of two metrics, m_i and m_j, as

$$z\left(\bar{m}_i, \bar{m}_j\right) = \begin{cases} 1, & \text{if } m_i \leq m_j \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Let $\bar{m}_i = (\bar{m}_{i,p}, \bar{m}_{i,p-1}, \cdots, \bar{m}_{i,0})$ and $\bar{m}_j = (\bar{m}_{j,p}, \bar{m}_{j,p-1}, \cdots, \bar{m}_{j,0})$ be the two's complement representation of the modulo metrics, each having a bit width of $p+1$, and let $\hat{m}_i = (\bar{m}_{i,p-1}, \cdots, \bar{m}_{i,0})$

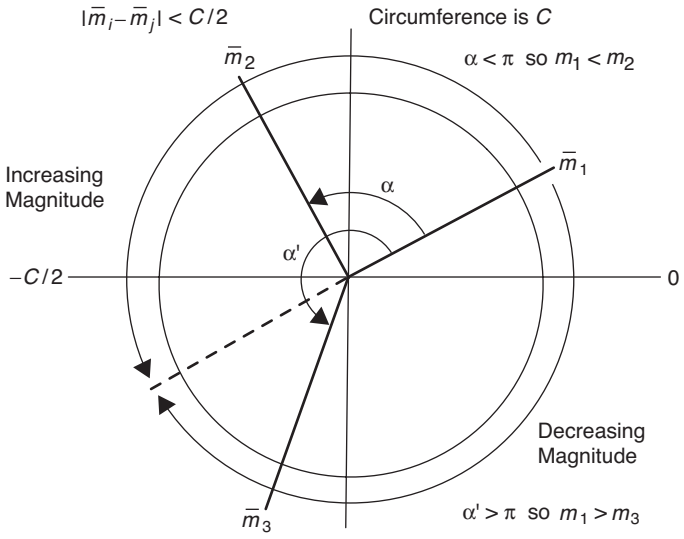

Fig. 12. The idea of modulo arithmetic. The reference metric is $\bar{m}_1$.

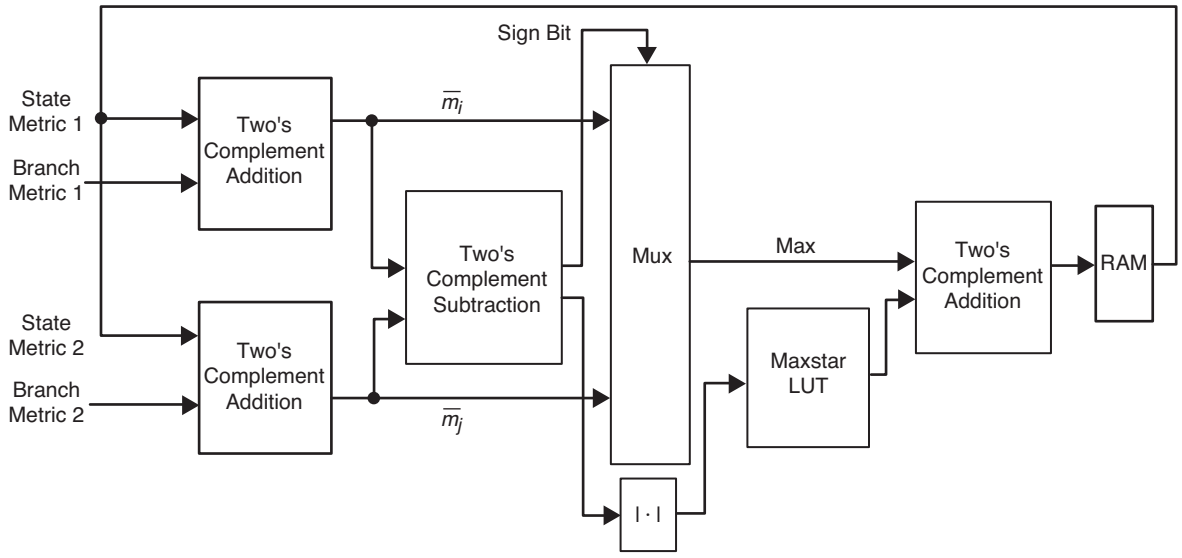

Fig. 13. Architecture for fast metric update using modulo arithmetic in the BCJR algorithm.

and $\hat{m}_j = (\bar{m}_{j,p-1}, \cdots, \bar{m}_{j,0})$ be their respective unsigned representations. Shung's modified comparison rule is

$$z\left(\bar{m}_i, \bar{m}_j\right) = \bar{m}_{i,p} \oplus \bar{m}_{j,p} \oplus y\left(\hat{m}_i, \hat{m}_j\right) \tag{11}$$

where $\oplus$ is the exclusive or (XOR) operator and

$$y\left(\hat{m}_i, \hat{m}_j\right) = \begin{cases} 1, & \text{if } \hat{m}_i \leq \hat{m}_j \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

is the unsigned comparison. Therefore, $z(\bar{m}_i, \bar{m}_j)$ is equal to $y(\hat{m}_i, \hat{m}_j)$ if $\bar{m}_i$ and $\bar{m}_j$ have the same sign, and $z(\bar{m}_i, \bar{m}_j)$ is the logical inverse of $y(\hat{m}_i, \hat{m}_j)$ otherwise. Since the operations $\bar{m}_{i,p} \oplus \bar{m}_{j,p}$ and $y(\hat{m}_i, \hat{m}_j)$ can be performed in parallel, the modified rule reduces a 1-bit comparison delay.

We incorporate Shung's modified rule into the modulo metric normalization procedure for the BCJR algorithm. The two's complement subtraction in Fig. 13 is a $(p+1)$-bit operation. We reduce this subtraction into a p-bit operation by concatenating the comparison rule of Eq. (11) with the result obtained by two's complement subtraction of $\bar{m}' = \hat{m}_i - \hat{m}_j$. In other words, the original subtraction result is equal to $(z(\bar{m}_i, \bar{m}_j), \bar{m}')$. Because computation of $z(\bar{m}_i, \bar{m}_j)$ and $\bar{m}'$ can occur in parallel, the 1-bit delay reduction offered by Shung's approach is preserved. The fast two's complement subtraction circuit is given in Fig. 14 and can be used to replace the subtraction module in Fig. 13 to achieve fast modulo normalization.

E. Partial Statistics

To reduce the channel likelihood storage requirements, we may discard the majority of the channel likelihoods and use partial statistics. This may be accomplished by processing only a subset consisting of the largest likelihoods during each symbol duration—the likelihoods corresponding to the PPM slots with the largest number of observed symbols. The observation of the remaining slots is set to the mean of a noise slot. In low background noise, a small subset may be chosen with negligible loss. More on this topic can be found in [16].

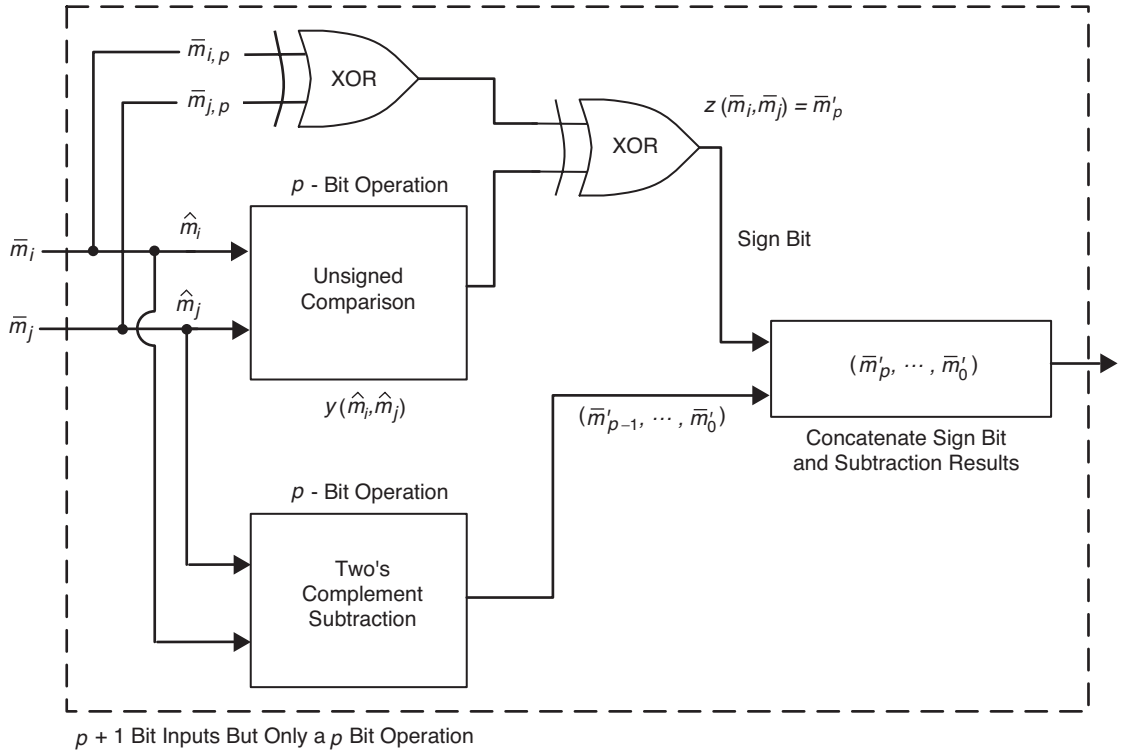

Fig. 14. A fast two's complement subtraction for modulo normalization. Inputs are $p+1$ bits but the circuit path is only p bits.

F. Window-Based Decoding Approach

We can partition the code trellis into distinct segments and decode these segments in parallel, therefore increasing the overall throughput. In iterative decoding, a CRC is often used as a stopping rule. While the concept of windowing is not new, we have not seen in the literature a description of an efficient CRC circuit that works with window-based turbo decoders. In Section VII, we will provide such a circuit realization.

The inner SCPPM trellis consists of $n = N/\log_2 M$ symbols or segments. The codeword length N is selected to be 15120 bits, and a practical PPM order M is 64. For this setting, the inner trellis will have 2520 segments. The outer trellis is a rate-1/2 code and therefore has $N/2$ or 7560 segments. We can partition the outer code by three and apply window-based BCJR to all three segments in parallel to obtain an overall increase in throughput by a factor of two, as seen in Fig. 15.

In simulation, we observed that we do not have to use a warm-up window to obtain a performance close to that of the original. That is, for the leftmost trellis segment, we can initialize the first state metric of the first stage to 0 (log-MAP decoding) and the remaining states to the smallest possible fixed-point value. The state metrics at the end of this trellis segment can be set to equal probability. For the middle trellis segment, we set the state metrics at the first stage and at the last stage to equal probability. For the rightmost trellis segment, we set the state metrics at the first stage to equal probability and terminate the trellis by assigning the first state metric of the last stage to 0 and the remaining state metrics to the smallest possible fixed-point value.

VI. The SCPPM Interleaver Design: Fast and Memory-Free

The interleaver design can affect the decoder threshold and error floor. Choosing a random interleaver permutation will generally lead to a desirable threshold, and the key to interleaver design becomes finding a permutation that will also lead to a low error floor. The SCPPM interleaver is characterized by a second-order polynomial $f(j) = \kappa j + \ell j^2$. We use choice selections of the parameters κ and ℓ to generate a permutation polynomial that not only exhibits a low error floor but also possesses a simple hardware implementation [17,18]. Section VIII.E provides a comparison of the SCPPM polynomial interleaver with the σ-random interleaver [19].

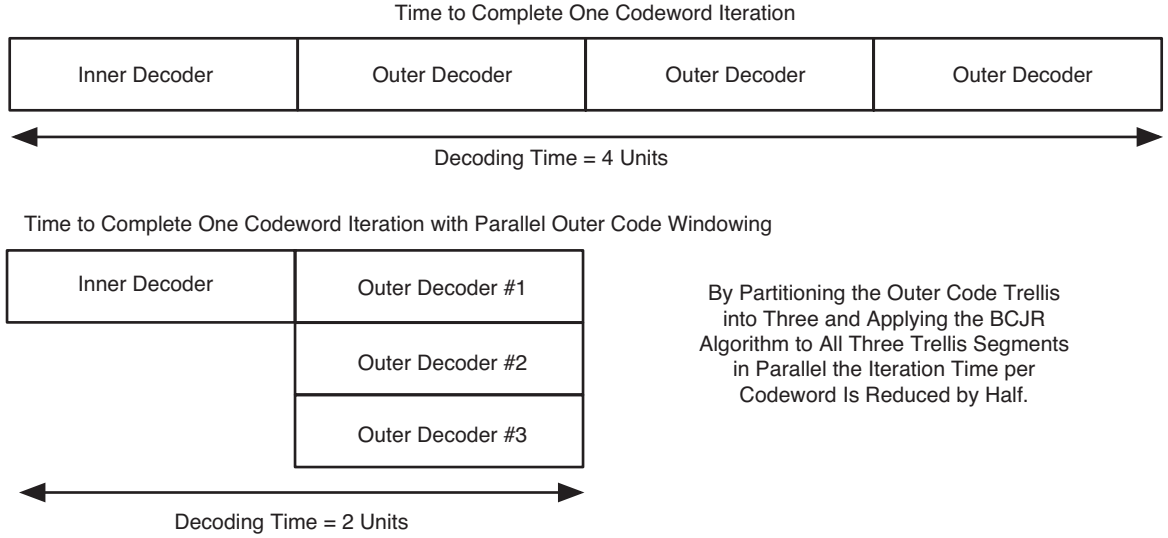

Fig. 15. Windowing increases throughput. For N = 15120 and 64 PPM, outer trellis is three times as long as the inner trellis.

The interleaver input bit position $f(j) \bmod N$ is mapped to output bit position j, i.e.,

$$\mathbf{x}_{f(j)} = \mathbf{a}_j$$

$$\mathbf{x}_i = \mathbf{a}_{f^{-1}(i)}$$

We show that mapping for the $(j+m)$th interleaver position can be expressed as a function of the current interleaver position j:

$$\begin{aligned} \big[f(j+m)\big]_N &= \big[\kappa(j+m) + \ell(j+m)^2\big]_N \\ &= \big[\big(\kappa j + \ell j^2\big) + \big(2m\ell j + m\left(\kappa + \ell m\right)\big)\big]_N \\ &= \big[f(j) + g(m,j)\big]_N \end{aligned} \tag{13}$$

where

$$g(m,j) = 2m\ell j + m(\kappa + \ell m) \tag{14}$$

and $[\cdot]_N$ is the "mod N" operation.

In our design, we assign $N = 15120 = 2^4 \cdot 3^3 \cdot 5 \cdot 7$. Candidate interleavers for this N are of the form $f(j) = \kappa j + 210\lambda j^2$ [17], where λ is a positive integer and κ does not have $2, 3, 5$, or 7 as a factor. Among this class we have observed good performance with the polynomial $f(j) = 11j + 210j^2$. An inverse polynomial is calculated in [2] and given as $f^{-1}(i) = 7331i + 7770i^2$. We use the inverse polynomial to implement the deinterleaver.

A. Interleaver Partitioning for One Clock Read/Write Access

For an M-order PPM modulation, the inner decoder processes a PPM symbol (or $\log_2 M$ bit LLRs) per trellis stage. A straightforward scheduling would be to read one LLR from the interleaver memory per clock. This approach incurs a long latency because the inner decoder would have to wait $\log_2 M$ clocks before proceeding to the next stage. To make interleaving more efficient, we design an approach that allows one clock read/write access. This approach also applies to the deinterleaver.

We illustrate our idea using the $M = 64$ SCPPM decoder with $N = 15120$. The interleaver memory is partitioned into $\log_2 64 = 6$ memory modules. This implementation can be easily adapted for codes with other PPM orders and parameters.

Each module is implemented using Xilinx dual-ported block random access memory (BRAM) as shown in Fig. 16. The input position into the inner decoder j is determined from the output position $f(j)$ of the outer decoder, that is $\mathrm{PaI}[j] \Leftarrow \mathrm{PxO}\left[\big[f(j)\big]_N\right]$. At each clock, the outer decoder produces two LLRs, and these are written in permuted order into the BRAMs simultaneously. The address permutation-to-memory location mapping for the interleaver is given in Table 1. The first column consists of the output position $\big[f(j)\big]_N$ of the outer decoder in sequential order. The second column consists of the corresponding input position j into the inner decoder. The third and fourth columns are the memory module index $(j \bmod 6)$ and address $(\lfloor j/6 \rfloor)$ in which the corresponding outer decoder output position is stored. The fifth column indicates the trellis stage, and the sixth column marks the BCJR window number (for the window-based SCPPM decoder). For example, the 221st LLR, starting from zero, produced by the outer

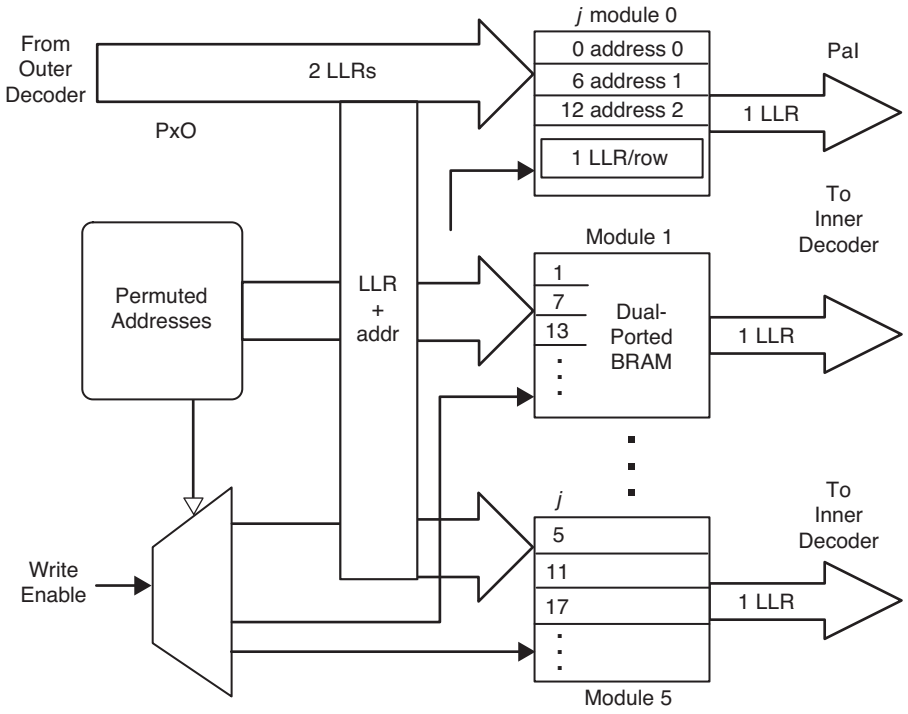

Fig. 16. Interleaver implementation. Permuted addresses can be obtained from lookup table or computed on-the-fly.

decoder corresponds to the first LLR input for the inner decoder. This LLR is stored in address zero of memory module one. This LLR is calculated at the 110th outer code trellis stage (0-7559) and belongs to the zeroth window segment (out of three).

The outer decoder writes to the interleaver BRAMs in permuted order using the mapping of Table 1. As we march down the table entries, we see that there will be no write conflicts at any time because the period of memory module writes is six and only two LLRs are produced by the outer decoder each clock. During interleaver reads, the inner decoder accesses the BRAM entries in sequential order. That is, at the first clock, the inner decoder reads the first entry (address 0) of each of the six memory modules and increases the address pointers by one. The six LLR reads correspond to PaI[0] through PaI[5] and are presented in a bold-faced font in Table 1. At the next clock, the inner decoder reads the second entry (address 1) of each memory module and again updates the address pointer. These six LLR reads correspond to PaI[6] through PaI[11], and so on.

The deinterleaver is implemented as one big chunk of memory, as illustrated in Fig. 17. The output LLRs generated by the inner decoder are written sequentially six at a time into one row of the dual-ported BRAM. The outer decoder then reads the LLRs in permuted order two at a time from the deinterleaver. The address permutation table for the deinterleaver is the same as that of the interleaver given in Table 1, with the exception that the header corresponds to that of the second row. For example, PxI[862], the 862nd LLR (starting from zero) input to the outer decoder should be read from the second column, zeroth row of the deinterleaver BRAM. The control logic reads the desired two rows and then selects the correct entry out of each row. One can see from the table that there are no read conflicts.

With the above interleaver and deinterleaver design, the LLRs produced or required by a stage of trellis decoding can be written to or read from memory in one clock cycle.

Table 1. Address permutation table for the interleaver (top header) and deinterleaver (bottom header).

$\mathbf{x}_{f(j)} : f(j)$	$\mathbf{a}_j : j$	Module	Address	Stage	Window
$\mathbf{x}_i : i$	$\mathbf{a}_{f^{-1}(i)} : f^{-1}(i)$	Column	Row	Stage	Window
0	**0**	**0**	**0**	**0**	**0**
1	15101	5	2516	0	0
2	382	4	63	1	0
3	1203	3	200	1	0
4	2444	2	407	2	0
5	4105	1	684	2	0
⋮	⋮	⋮	⋮	⋮	⋮
221	**1**	**1**	**0**	**110**	**0**
⋮	⋮	⋮	⋮	⋮	⋮
862	**2**	**2**	**0**	**431**	**0**
⋮	⋮	⋮	⋮	⋮	⋮
5040	10080	0	1680	2520	1
5041	10061	5	1676	2520	1
⋮	⋮	⋮	⋮	⋮	⋮
10080	5040	0	840	5040	2
10081	5021	5	836	5040	2
⋮	⋮	⋮	⋮	⋮	⋮
15119	439	1	73	7559	2

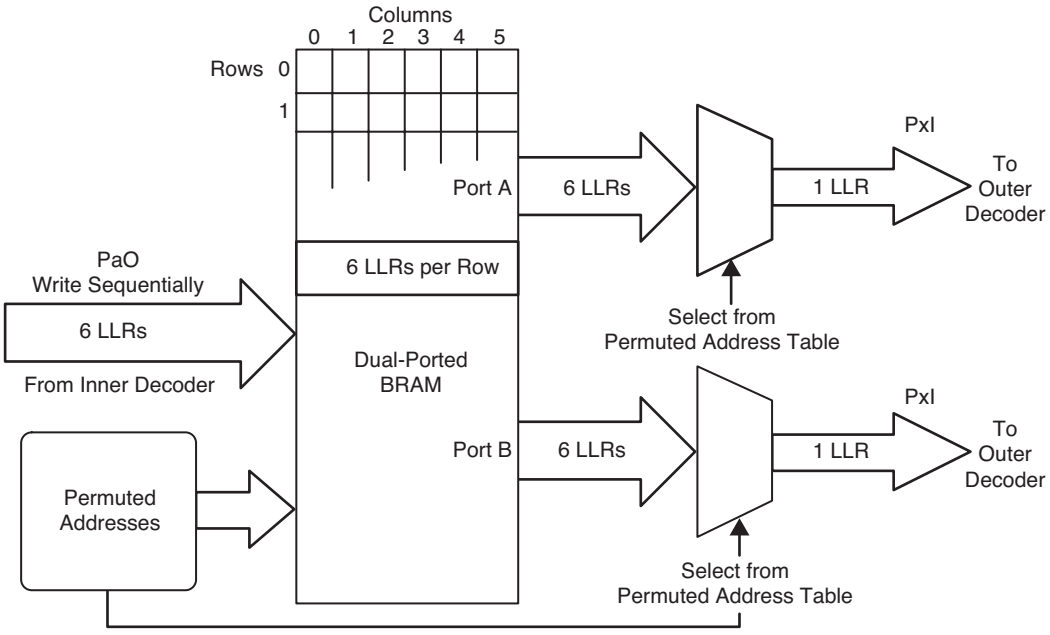

Fig. 17. Deinterleaver implementation. Permuted addresses can be obtained from a lookup or computed on-the-fly.

B. No Need to Store Interleaver Mappings: An Algorithmic Implementation

We can avoid the need to store Table 1 in memory by computing the memory module and address for a specific interleaved position on-the-fly. The interleaver is partitioned into C distinct memory blocks, each with $n = N/C$ entries for fast read and write access. Each interleaver position $\big[f(j)\big]_N$, for $j = [0, 1, \cdots, N-1]$, is mapped to a corresponding index pair $\big(r_{f(j)}, q_{f(j)}\big)$, where $r_{f(j)} \triangleq \big[f(j)\big]_C$ is the index into one of the C memory modules and $q_{f(j)} = \big[\, \lfloor f(j)/C \rfloor \,\big]_n$ is the index into one of the n address entries in each module. With windowing, the inner and outer decoder exchanges $C = \log_2 M$ bit LLRs per decoding stage. Because the modulo and division operations are costly to implement in hardware, we describe a procedure that calculates the interleaver indexing pair for the set of C bit LLRs desired by the current stage based on the set of C indexing pairs computed in the previous stage. We begin with a proposition.

Proposition 1. If $C \mid N$, then $\big[\big[f\,(j)\big]_N\big]_C$ is equivalent to $\big[f\,(j)\big]_C$.

Proof. A nonnegative number f modulo C can be obtained by continuously subtracting C from f until f becomes less than C. If $C \mid N$, the number f modulo N can be obtained by subtracting $n = N/C$ multiples of C from f. Therefore, $\big[\big[f(j)\big]_N\big]_C$ is equivalent to $\big[f(j)\big]_C$. ❒

Step 1. Initialization. We assign the constant $\Delta_q = [2C\ell]_n$, define $g(l) \triangleq g(C, l)$, and set the initial modulus values using Eqs. (13) and (14) to

$$r_{f(0)} = \big[f(0)\big]_C, \cdots, r_{f(C-1)} = \big[f(C-1)\big]_C \tag{15}$$

$$r_{g(0)} = \big[g(C,0)\big]_C, \cdots, r_{g(C-1)} = \big[g(C,C-1)\big]_C \tag{16}$$

as well as the initial quotient values

$$q_{f(0)} = \left[\left\lfloor \frac{f(0)}{C} \right\rfloor\right]_n, \cdots, q_{f(C-1)} = \left[\left\lfloor \frac{f(C-1)}{C} \right\rfloor\right]_n \tag{17}$$

$$q_{g(0)} = \left[\left\lfloor \frac{g(C,0)}{C} \right\rfloor\right]_n, \cdots, q_{g(C-1)} = \left[\left\lfloor \frac{g(C,C-1)}{C} \right\rfloor\right]_n \tag{18}$$

Step 2. Loop for stage $= 1 : n-1$ **and LLR bit** $i = 0 : C-1$ **within the stage.** Note that each i update is implemented by an individual circuit. Therefore, we have C circuits working in parallel, each calculating the interleaver indexing pair for each of the C LLRs needed by decoding of the inner code. First, we expand

$$\begin{aligned} g(C, i+C) &= 2c\ell(i+C) + C(\kappa + \ell C) \\ &= g(C,i) + 2C^2\ell \end{aligned} \tag{19}$$

We update the modulus by applying Eq. (13) as

$$
\begin{aligned}
r_{f(\text{stage}\cdot C+i)} &= \Big[f\big(\overbrace{(\text{stage}-1)\cdot C+i}^{j}+\overbrace{C}^{m}\big)\Big]_C \\
&= \big[f\big((\text{stage}-1)\cdot C+i\big)+g\big((\text{stage}-1)\cdot C+i\big)\big]_C \\
&= r_{f((\text{stage}-1)C+i)} \qquad (20)
\end{aligned}
$$

because for all stages and applying Eq. (19)

$$
\begin{aligned}
r_{g(\text{stage}\cdot C+i)} &= \big[g\big((\text{stage}-1)\cdot C+i\big)+2C^2\ell\big]_C \\
&= 0 \qquad (21)
\end{aligned}
$$

We express the functions

$$
f\big((\text{stage}-1)\cdot C+i\big) = q_f C + r_f \qquad (22)
$$

and

$$
g\big((\text{stage}-1)\cdot C+i\big) = q_g C + r_g \qquad (23)
$$

We follow by updating the quotient for f as

$$
\begin{aligned}
q_{f(\text{stage}\cdot C+i)} &= \left[\left\lfloor \frac{q_f C + r_f + q_g C + r_g}{C} \right\rfloor\right]_n \\
&= \left[q_f + q_g + \left\lfloor \frac{r_f + r_g}{C} \right\rfloor\right]_n \\
&= \big[q_{f((\text{stage}-1)C+i)} + q_{g((\text{stage}-1)C+i)}\big]_n \qquad (24)
\end{aligned}
$$

because from Eq. (20) $r_f < C$ and from Eq. (21) $r_g = 0$. We can then update the quotient for g as

$$
\begin{aligned}
q_{g(\text{stage}\cdot C+i)} &= \left[\left\lfloor \frac{g\left((\text{stage}-1)\cdot C+i\right)+2C^2\ell}{C} \right\rfloor\right]_n \\
&= \left[\left\lfloor \frac{q_g C + r_g}{C} \right\rfloor + 2C\ell\right]_n \\
&= \big[q_{g((\text{stage}-1)\cdot C+i)} + \Delta_q\big]_n \qquad (25)
\end{aligned}
$$

The memory module for the interleaver position $f\left(\text{stage}\cdot C+i\right)$ is then $r_{f(\text{stage}\cdot C+i)}$, and the address entry is $q_{f(\text{stage}\cdot C+i)}$.

C. Circuit Description for the Algorithmic Interleaver

The derivations of Eqs. (20) and (21) indicate that the memory module index for each bit LLR and each stage stays the same throughout the trellis. Plugging in parameters to the initial values of Eq. (15) will show that the memory module index has a period C. This observation is confirmed by Table 1. In the table, the memory module indices take on the values $[0, C-1, C-2, \cdots, 2, 1]$ for every stage, and this pattern repeats for all stages. Consequently, we need to calculate only the address entry for each bit LLR in each stage. A circuit that implements Eqs. (24) and (25) to compute the address entry of each LLR per trellis stage is given in Fig. 18. Note that there will be C such circuits, one for each LLR, working in parallel.

This algorithmic interleaver removes the need to store Table 1. Implementation requires only a small number of gates. This memory-saving benefit is even more evident in multiple decoder instantiations on one FPGA because many copies of the same table need not be stored. Some idea of the actual savings provided by the algorithmic interleaver is found in Section VIII.

VII. A CRC Circuit for Window-Based Turbo Decoders

A straightforward hardware implementation of a cyclic redundancy check (CRC) is simply a linear feedback shift register (LFSR). A block of information bits and the associated CRC check bits are shifted into the LFSR circuit one bit at a time. After the entire block is input to the circuit, the state of the registers indicates whether the CRC passed or not. A CRC can be used together with iterative turbo decoding to flag codeword errors or to stop decoding iterations. To increase the throughput of turbo decoding, the code trellis can be partitioned into distinct windows, and multiple decoders can be applied to these windowed trellis segments in parallel. In windowed-based turbo decoding, the bits to be input to the CRC will be generated in parallel, more than one at a time. Therefore, the serial input CRC circuit needs to be modified to handle this parallelism.

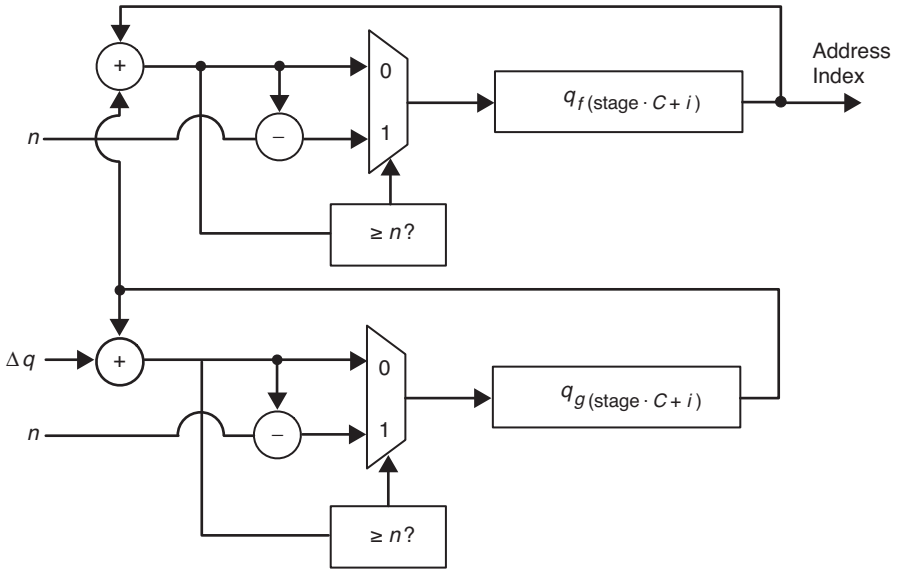

Fig. 18. A circuit that computes the address into the memory module for the *i*th LLR in each stage, where $i = [0, \cdots, C-1]$. There will be C instances of this circuit, one for each LLR.

A. Polynomial Description of CRCs

Let us write a length-k binary message block $\mathbf{m} = (m_{k-1}, m_{k-2}, \cdots, m_0)$, which is to be protected by a CRC, in polynomial form:

$$m(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_0 \tag{26}$$

Let the length-n CRC-protected codeword be $\mathbf{c} = (c_{n-1}, c_{n-2}, \cdots, c_0)$ or

$$c(x) = c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \cdots + c_0 \tag{27}$$

and the CRC generator be

$$g(x) = g_{n-k}x^{n-k} + \cdots + g_0 \tag{28}$$

The CRC polynomial $r\,(x)$ is calculated by first shifting the message polynomial left by $n-k$ positions and then by taking the modulo $g\,(x)$ operation

$$r(x) = R_{g(x)}\big[m(x) \cdot x^{n-k}\big] \tag{29}$$

where $\deg\big[r(x)\big] < n-k$. The codeword block can also be written as

$$\begin{aligned} \mathbf{c} &= [\mathbf{m}, 0, \cdots, 0] \oplus \mathbf{r} \\ &= \left[m_{k-1}, \cdots, m_0, \quad \underbrace{0, \cdots, 0}_{n-k}\right] \oplus [r_{n-k-1}, \cdots, r_0] \end{aligned} \tag{30}$$

($\oplus$ is the binary XOR operations) or

$$c(x) = m(x) \cdot x^{n-k} + r(x) \tag{31}$$

To verify the CRC of a codeword block $\hat{c}\,(x) = c\,(x) + e(x)$ that may be corrupted by an error polynomial $e(x)$, we calculate

$$\begin{aligned} R_{g(x)}\big[\hat{c}\,(x)\big] &= R_{g(x)}\big[m(x) \cdot x^{n-k} + r(x) + e(x)\big] \\ &= R_{g(x)}\left[R_{g(x)}\big[m(x) \cdot x^{n-k}\big] + R_{g(x)}\big[r(x)\big] + R_{g(x)}\big[e(x)\big]\right] \\ &= r(x) + r(x) + R_{g(x)}\big[e(x)\big] \\ &= R_{g(x)}\big[e(x)\big] \end{aligned} \tag{32}$$

Therefore, if the remainder is zero, the CRC passes and the error polynomial is zero. If the remainder is nonzero, then the codeword is corrupted. Note that we won't be able to construct the error polynomial $e(x)$ from the CRC remainder $R_{g(x)}\big[e(x)\big]$.

B. Hardware Description of CRC Checks

A CRC is simply a modulo operation and can be implemented by an LFSR for dividing polynomials. The circuit for multiplying by a polynomial $h(x)$ and dividing by a polynomial $g(x)$, each with degree up to ℓ, is given in Fig. 19. For division only, simply set $h(x) = 1$ ($h_0 = 1$; every other coefficient to 0). After the entire codeword is shifted into the circuit, the quotient of the division operation is given by the bits that are shifted out, and the remainder is given by the register state. More information on LFSRs can be found in [20, Linear Switching Circuits].

C. CRC Circuit for Windowed-Based Turbo Decoding

In windowed-based turbo decoding, the output bit streams to be fed into the CRC are generated in parallel, as seen in Fig. 20. We describe how a CRC circuit can be modified to handle this parallelism. Let the code trellis be partitioned into j distinct windows. The codeword polynomial can be written as

$$c\,(x) = c_1\,(x)\,x^{s_1} + c_2\,(x)\,x^{s_2} + \cdots + c_j\,(x) \tag{33}$$

We can then write the check polynomial as

$$\begin{aligned} R_{g(x)}\big[c(x)\big] &= R_{g(x)}\big[c_1(x)x^{s_1} + c_2(x)x^{s_2} + \cdots + c_j(x)\big] \\ &= R_{g(x)}\Big[R_{g(x)}\big[c_1(x)x^{s_1}\big] + R_{g(x)}\big[c_2(x)x^{s_2}\big] + \cdots + R_{g(x)}\big[c_j(x)\big]\Big] \\ &= R_{g(x)}\Big[R_{g(x)}\big[c_1(x)\kappa_1(x)\big] + R_{g(x)}\big[c_2(x)\kappa_2(x)\big] + \cdots + R_{g(x)}\big[c_j(x)\big]\Big] \end{aligned} \tag{34}$$

where $\kappa_i = R_{g(x)}[x^{s_i}]$, $i = 1, 2, \cdots, j-1$, and each $\kappa_i(x)$ can be precalculated. The CRC LFSR circuit for the window-based decoder will consist of both feedforward and feedback tap connections. The feedforward taps are given by the XOR of $\kappa_i(x)$'s, and the feedback taps are given by the generator $g(x)$.

D. A CRC Circuit for the Window-Based SCPPM Decoder

A practical realization of the SCPPM code scheme is to use PPM order 64. The SCPPM decoder in our implementation is windowed by three, as detailed in Section V.F, to double the overall throughput. With the outer code trellis having 7560 stages, each windowed-by-three segment has 2520 stages. We

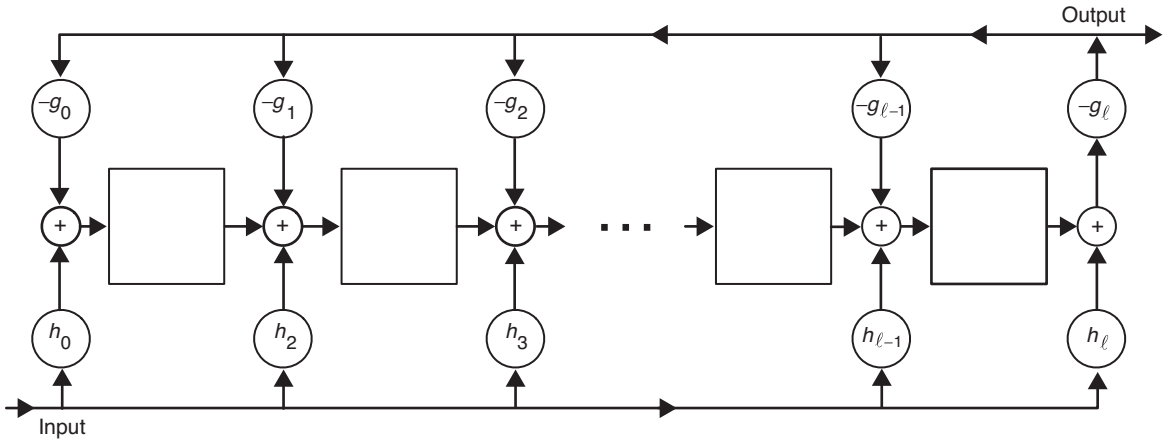

Fig. 19. A circuit for multiplying by $h(x)$ and dividing by $g(x)$.

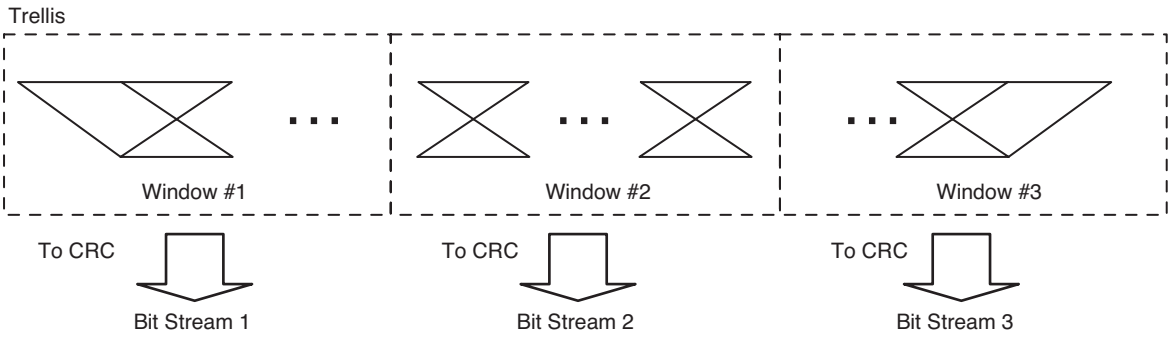

Fig. 20. A trellis windowed by three leads to three simultaneous decoded bit streams.

use a 22-bit CRC with generator $g(x) = x^{22} + x^5 + x^4 + x^3 + 1$ to check the output of the windowed SCPPM decoder. The CRC indicates whether a correct codeword decision is reached and can be used to stop the iteration process. Using techniques presented in this section, we precompute the polynomials $\kappa_1(x) = R_{g(x)}\left[x^{5040}\right]$ and $\kappa_2(x) = R_{g(x)}\left[x^{2520}\right]$ and generate three circuits (shown in Fig. 21) to check the output bit stream of each window. We can optimize and consolidate the three circuits into one by XORing the three output bit streams according to the feedforward taps before inputting to the CRC circuit.

VIII. Results for Decoder Implementation and Performance

The SCPPM decoder for PPM order $M = 64$ is currently implemented on a Xilinx Virtex II-8000 FPGA part, speed grade 4 (XC2V8000-4), which sits on a Nallatech BenDATA-WS board. The memory requirement is reduced by taking only the top 8 channel LLRs as decoder input. The channel LLRs input to the decoder are quantized to 8 bits, 5 for dynamic range and 3 for precision.

We have implemented three versions of the decoder. The first is the log-MAP decoder with normalization and clipping circuits for the state metrics. The backward recursion state metrics $\bar{\beta}$'s are clipped to 8 bits before being stored into RAMs. The forward recursion state metrics $\bar{\alpha}$'s are calculated as needed and are not stored. The remaining variables in the data path are allowed to grow and are not stored.

The second is the max log-MAP decoder with modulo normalization. The $\bar{\beta}$'s are allowed to grow in dynamic range up to 16 bits (plus a 3-bit precision for a total of 19 bits) before being stored into RAMs. Again, the $\bar{\alpha}$'s are calculated as needed and are not stored. All other metrics are allowed to grow in width and are not stored.

The third is the window-based max log-MAP decoder. The outer code trellis is partitioned into three.

We had the opportunity to complete only the place and route for a fourth variation of the decoder, the "maxstar top 2" implementation, and did not get a chance to finish the wrapper around the decoder. But we did produce a bit-exact software of the maxstar top 2 decoder and used it to generate accurate simulation results.

A. Resource Utilization

The total FPGA resource utilization as well as a breakdown by modules for each of the decoders are given in Tables 2 through 4. The percentage of resources used by the inner decoder, outer decoder, and the remaining blocks equal the total utilization. Blocks other than the decoder modules that consume resources are the circuitries and memories instantiated for the FPGA interface.

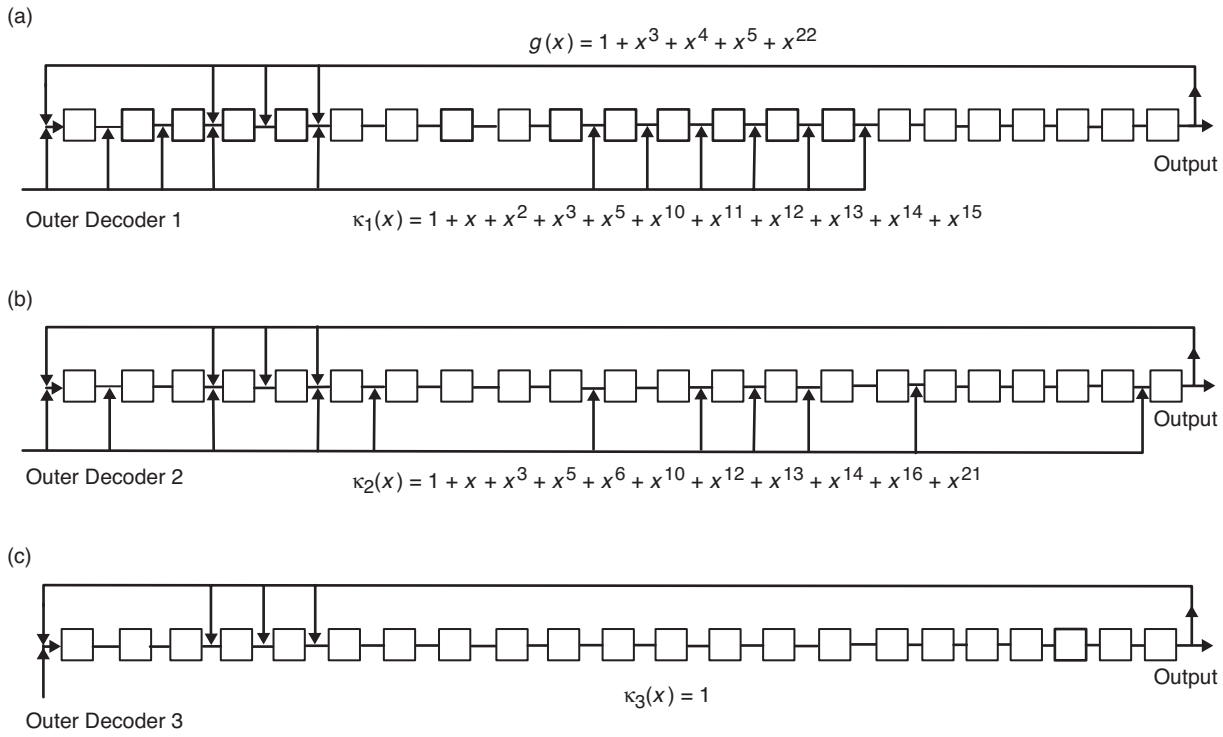

Fig. 21. CRC circuits for the polynomial $g(x) = x^{22} + x^5 + x^4 + x^3 + 1$ and the SCPPM windowed by three decoder: (a) outer decoder 1, (b) outer decoder 2, and (c) outer decoder 3.

Table 2. The log-MAP SCPPM decoder on the Virtex-II 8000 FPGA.

	Used/total	Utilization, percent	Inner, percent	Outer, percent	Others, percent
BRAM	101/168	60	19	9	32
Slices	30174/46592	64	52	6	6

Table 3. The max log-MAP SCPPM decoder on the Virtex-II 8000 FPGA.

	Used/total	Utilization, percent	Inner, percent	Outer, percent	Others, percent
BRAM	86/168	51	19	9	23
Slices	19927/46592	42	31	5	6

Table 4. The max log-MAP window-based SCPPM decoder on the Virtex-II 8000 FPGA.

	Used/total	Utilization, percent	Inner, percent	Outer, percent	Others, percent
BRAM	134/168	80	19	30	31
Slices	24587/46592	53	32	15	6

The $\overset{*}{\max}$ lookup tables (LUTs) for the log-MAP decoder are realized as read-only memories (ROMs). The channel symbol memory and state metric storage memory are all implemented using Xilinx internal, dual-ported block RAMs (BRAMs). In the log-MAP decoder, we store the interleaver mappings as LUTs on BRAMs. This allows us to compare the memory savings in going to an algorithmic interleaver. For the max log-MAP version, we avoid storing the interleaver mapping and use the algorithmic approach. Comparing the two BRAM utilizations, we see that a 9 percent BRAM savings is achieved on the Virtex-II FPGA using the algorithmic interleaver.

Table 5 directly compares the logic usage in the log-MAP, max log-MAP, and maxstar top 2 log-MAP implementations. The log-MAP and max log-MAP decoders trade off logic and performance. If both metrics are deemed to be important, we can compromise and use the hybrid maxstar top 2 log-MAP decoder. The maximum clock rates and throughput based on 7 average iterations for three decoder designs are given in Table 6.

B. Error-Rate Performance

The decoder performance is shown in Fig. 22. The frame-loss rate (FLR) is plotted versus n_s, the average signal photons per pulse slot in decibels. Each frame is a codeword of $k = 7560$ information bits. A frame loss is declared when the decoder decision cannot converge to the correct codeword in the maximum number of allowed iterations, which is set at 32. Out of the 7560 bits, 2 bits are used to terminate the trellis and 22 bits are used for CRC. The CRC polynomial is $x^{22} + x^5 + x^4 + x^3 + 1$ and has an undetected word-error probability of approximately $7 \cdot 2^{-22} = 1.67 \times 10^{-6}$, assuming 7 average iterations. To reduce the undetected rate, the decoder runs a minimum number of iterations first before validating the CRC. In doing so, the undetected probability is lowered to roughly the product of the frame-loss rate and 1.67×10^{-6}, a very small value.

We make the following observations from the performance plot. Fixed-point implementation (circle line) has a 0.1-dB loss compared to the floating-point decoder (dashed line). Clipping and normalization of the state metrics leads to a floor at 10^{-5}. The max log-MAP decoder (square line) has a 0.6-dB loss compared to log-MAP decoding (circle line). The max log-MAP decoder with a scaling of the extrinsic information by 0.5 (diamond line) recovers 0.4 dB out of the 0.6 dB lost. Note that only the extrinsic information at the output of the inner decoder is scaled by a factor between 0 and 1. The extrinsic information at the output of the outer decoder is untouched. The clipping and normalization floor is lowered by using modulo arithmetic.

Also notice that, using the maxstar top 2 (triangle line) circuit in the inner decoder, the log-MAP outer decoder, and a scaling of inner decoder extrinsic information by 0.625, we are able to recover another 0.075 dB in signal energy. A 0.5 scale factor can be implemented in hardware by simply a right shift by 1. A scaling of 0.625 is the sum of a right shift by 1 and right shift by 3.

Table 5. Resource utilization of the Xilinx Virtex-II 8000 FPGA by the three decoding approaches.

	log-MAP, percent	maxstar top 2 log-MAP, percent	max log-MAP, percent
Slice utilization	64	54	42

Table 6. Maximum clock rate and throughput for the two SCPPM decoder designs on the Xilinx Virtex-II 8000 FGA.

	log-MAP	max log-MAP	Windowed max log-MAP
Maximum clock, MHz	23	63	63
Throughput, Mbps	1.23	3.36	6.72

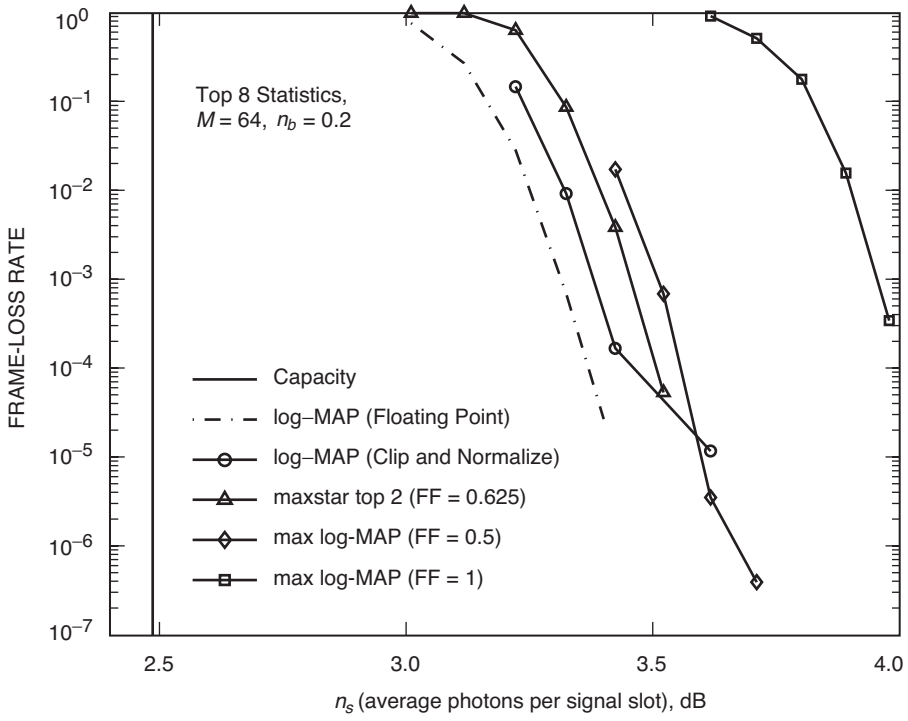

Fig. 22. SCPPM decoder performance on the Poisson channel under a nominal deep-space mission scenario.

C. End-to-End Demonstration

We have demonstrated an end-to-end optical communications system in our laboratory using the SCPPM scheme. The transmitter employs a 1064-nm-wavelength laser to modulate a stream of SCPPM-encoded information into PPM pulses. The pulses are then delivered over a fiber-optic channel. At the receiving end, a photon-counting detector is used, and the receiver assembly converts the photon counts into LLRs for the SCPPM FPGA decoder described in this section. The results of the experimental runs at various operating points are plotted in Fig. 23. There are two experimental runs, one at 4 Mbps and the other at 6 Mbps. These two curves are compared to a curve generated by using a software-simulated Poisson channel and the stand-alone FPGA decoder. We see that the experimental curves match very closely to the stand-alone FPGA result. The system runs at an average of 7 iterations. The end-to-end performance is within 1.4 dB of channel capacity.

D. Performance Comparison Versus Reed–Solomon–PPM Scheme

A legacy ECC used in many previous and current NASA missions is the Reed–Solomon code. In Fig. 24 we compare the Reed–Solomon PPM (RS-PPM)-coded scheme versus the SCPPM-coded scheme and show that, in a nominal mission scenario, SCPPM outperforms RS-PPM by 3 dB. The results are generated using software simulation. To match the code rates, we use the SCPPM code parameters $(N, K) = (16398, 8199)$ and the RS-PPM code parameters $(4085, 2047)$. We choose 64 PPM as a reasonable order and an average background count of 0.2 photons per slot.

E. Performance Comparison of Different Interleavers

The interleaver design affects the error-rate performance and error floor of SCPPM. Through simulation, we show that the SCPPM permutation polynomial interleaver performs as well as a σ-random interleaver and has no observable error floor. The error-rate curves are plotted in Fig. 25. We see that for both the word-error rate (WER) and bit-error rate (BER) the two interleavers produced almost identical decoder performances. For deep-space missions, where minimum WER floor requirements are generally that of 10^{-4}, the two interleavers meet the specifications.

F. Path to 50 Mbps and Beyond

We achieved a 6.72-Mbps decoder on a single Xilinx Virtex-II FPGA. Currently, Xilinx has available the Virtex-II Pro FPGA part that is manufactured with a smaller micron process and features more BRAMs. We have completed a place and route of our fastest design on the Virtex-II Pro. Results indicate that the SCPPM decoder can deliver 8 Mbps at 7 average iterations. We can add another stage of parallelism to our design so that the inner decoder and outer decoder can work on two codewords simultaneously and are not idle at any time. Doing so doubles our throughput to 16 Mbps per FPGA. Moreover, we can realize multiple instances of our decoder on the Nallatech BenNuey-4E PCI board that has slots for three daughters, each capable of hosting two Virtex-II Pro FPGAs. This migration path leads to a 96-Mbps SCPPM decoder that is fit for deep-space optical communications. We can further increase the throughput to hundreds of megabits and beyond by implementing a lower-order SCPPM decoder, such as 16-PPM, for terrestrial applications where a smaller PPM order actually achieves higher capacity due to the shorter distance between the transmitter and receiver.

IX. Conclusion

The serially concatenated pulse-position modulation (SCPPM) capacity-approaching code was designed by NASA to support deep-space optical communications at 50 Mbps and beyond. The structure of the SCPPM trellis makes direct application of conventional turbo decoding very inefficient. We therefore introduced the following new techniques that optimize the overall decoder throughput and performance:

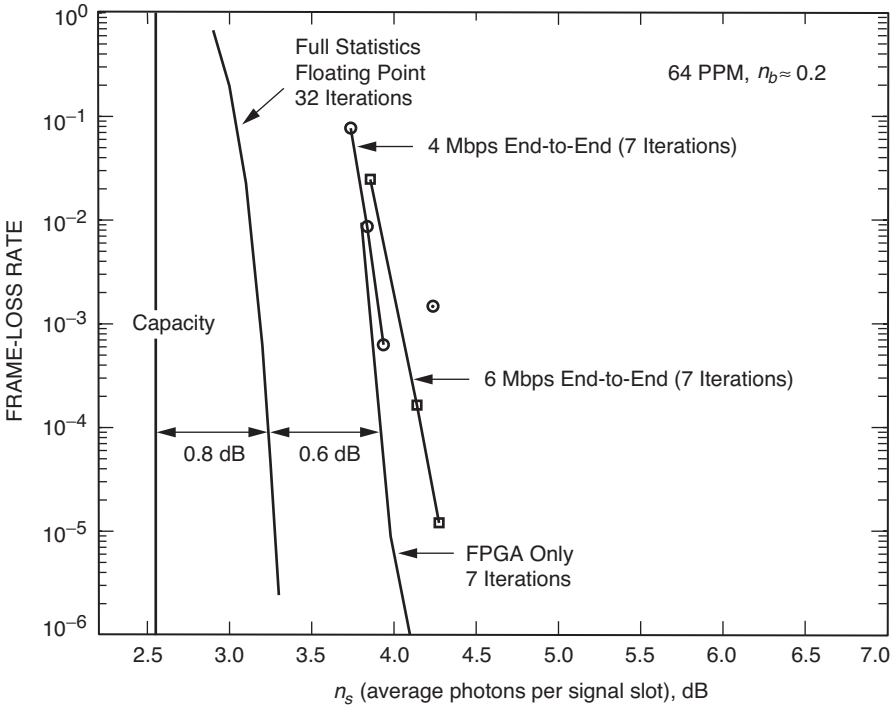

Fig. 23. End-to-end system demonstration of the SCPPM scheme over the optical channel.

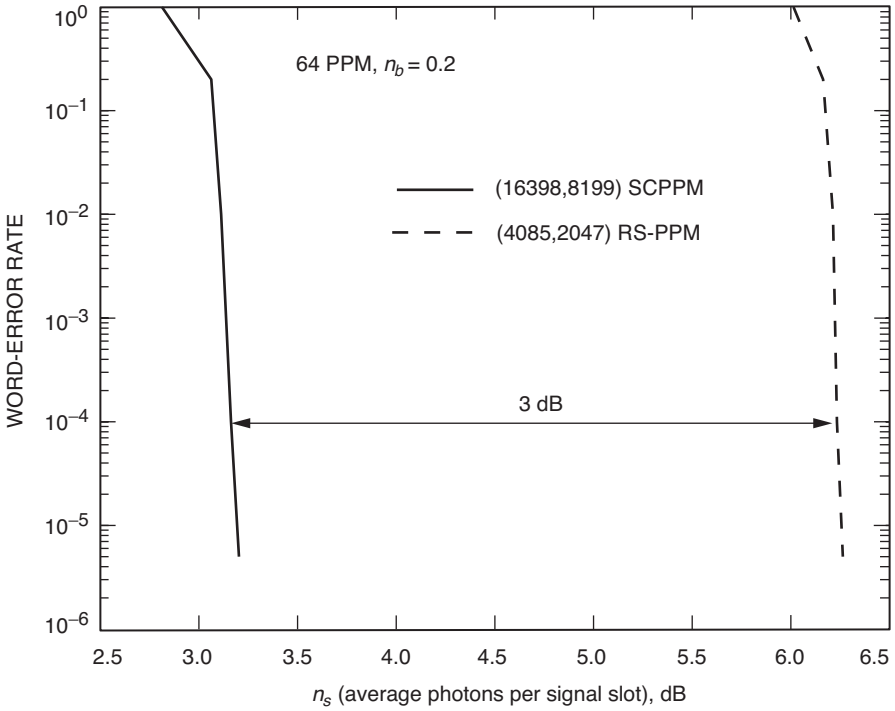

Fig. 24. Comparison of SCPPM versus RS-PPM under nominal deep-space mission scenario.

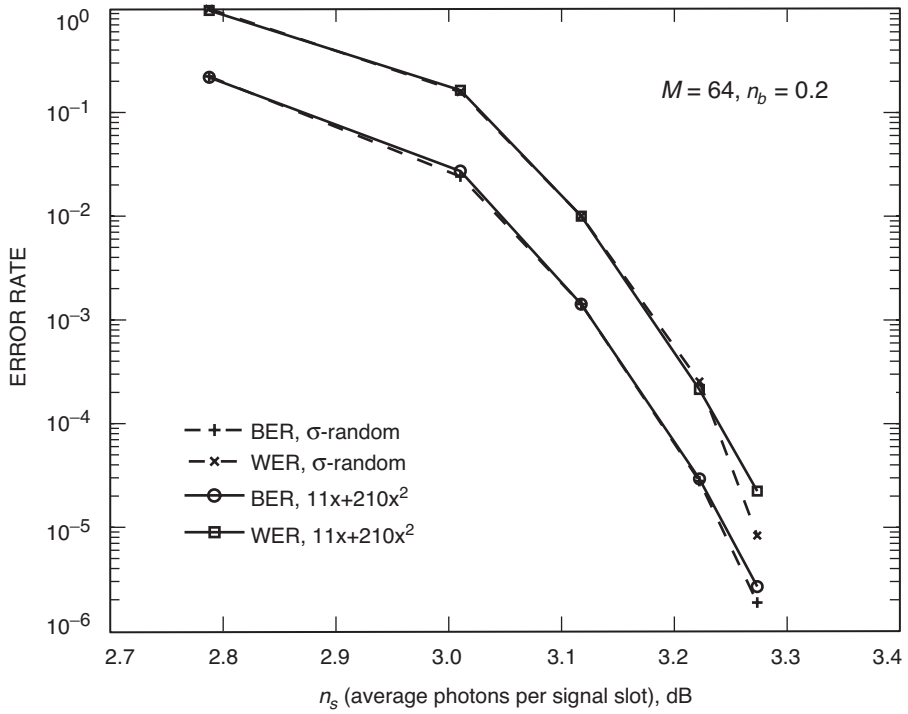

Fig. 25. Comparison of the SCPPM polynomial interleaver versus the σ-random interleaver.

a simplified Super Gamma computation to handle parallel trellis edges, a hybrid "maxstar top 2" circuit to manage memory and error-rate performance, and a modulo arithmetic circuit with a short path delay to avoid clipping and normalization.

The SCPPM trellis is also windowed to further increase data rate. We designed a compact CRC circuit that works with window-based decoders and incorporated a CRC to stop the decoding iterations. Moreover, we presented a fast and memory-free algorithmic interleaver implementation. Our algorithmic interleaver performs as well as any carefully designed random interleaver and exhibits a very low error floor.

To compare these techniques, we implemented three variations of the SCPPM decoder on a Xilinx Virtex-II 8000 FPGA and summarized their trade-offs. Through hardware simulation, we demonstrated that the SCPPM-coded scheme can perform within 1 dB of capacity in a nominal mission condition. Our techniques are applicable to any modulation and code scheme that has a high peak-to-average-power ratio designed to fit the requirements of optical communications.

We summarize the hardware implementation trade space in Table 7. For free-space optical communications, this chart allows a potential designer to select the best modulation and coding features that would meet a desirable performance requirement and fit onto an affordable FPGA real estate.

Table 7. Decoder implementation trade space.

Description	Options	Data rate	Performance loss, dB	Complexity	Memory use
Slot statistics from receiver	Top 8	—	0.2	—	1x
	Full M	—	0	—	4x
Algorithm	max log-MAP	1.7x	0.2	0.74x	—
	log-MAP	1x	0	1x	—
Arithmetic	Modulo	1.7x	—	0.9x	—
	Clip and normalize	1x	—	1x	—
Windowing	Outer trellis	2x	—	1.1x	1.08x
	None	1x	—	1x	1x
Iterations	Variable/CRC stopping rule	>1x	0	—	—
	Fixed	1x	$\geq$0	—	—

References

[1] J. R. Pierce, "Optical Channels: Practical Limits with Photon Counting," *IEEE Transactions on Communications*, vol. 26, pp. 1819–1821, December 1978.

[2] B. Moision and J. Hamkins, "Coded Modulation for the Deep-Space Optical Channel: Serially Concatenated Pulse-Position Modulation," *The Interplanetary Network Progress Report*, vol. 42-161, Jet Propulsion Laboratory, Pasadena, California, pp. 1–25, May 15, 2005.
http://ipnpr/progress_report/42-161/161T.pdf

[3] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A Soft-Input Soft-Output Maximum A Posteriori (MAP) Module to Decode Parallel and Serial Concatenated Codes," *The Telecommunications and Data Acquisition Progress Report 42-127, July–September 1996*, Jet Propulsion Laboratory, Pasadena, California, pp. 1–20, November 15, 1996.
http://ipnpr/progress_report/42-127/127H.pdf

[4] H. Hemmati, ed., *Deep Space Optical Communications*, Hoboken, New Jersey: John Wiley & Sons Inc., 2006.

[5] K. J. Quirk and L. B. Milstein, "Optical PPM with Sample Decision Photon Counting," *Proceedings of the IEEE Global Telecommunications Conference*, St. Louis, Missouri, pp. 148–151, December 2005.

[6] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Transactions on Information Theory*, vol. 20, pp. 284–287, March 1974.

[7] W. E. Ryan, *A Turbo Code Tutorial*, 1997.
http://www.ece.arizona.edu/~ryan/publications/turbo2c.pdf

[8] T. V. Souvignier, *Turbo Decoding for Partial Response Channels*, Ph.D. thesis, University of California, San Diego, 1999.

[9] A. J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 260–264, February 1998.

[10] G. Montorsi and S. Benedetto, "Design of Fixed Point Iterative Decoders for Concatenated Codes with Interleavers," *IEEE Journal on Selected Areas in Communications*, vol. 19, pp. 871–882, May 2001.

[11] P. H. Wu and S. M. Pisuk, "Implementation of a Low Complexity, Low Power, Integer-Based Turbo Decoder," *Proceedings of the IEEE Global Telecommunications Conference*, vol. 2, San Antonio, Texas, pp. 946–951, 2001.

[12] J. Vogt and A. Finger, "Improving the max-log-MAP Turbo Decoder," *Electronic Letters*, vol. 36, pp. 1937–1939, November 2000.

[13] M. Barsoum and B. Moision, *Method and Apparatus for Fast Digital Turbo Decoding for Trellises with Parallel Edges*, JPL Novel Technical Report no. 4123, Jet Propulsion Laboratory, Pasadena, California, July 2004.

[14] A. P. Hekstra, "An Alternative to Metric Rescaling in Viterbi-Decoders," *IEEE Transactions on Communications*, vol. 37, pp. 1220–1222, November 1989.

[15] C. Shung, P. Siegel, G. Ungerboeck, and H. K. Thapar, "VLSI Architectures for Metric Normalization in the Viterbi Algorithm," *Proceedings of the IEEE International Conference on Communications*, vol. 4, Atlanta, Georgia, pp. 1723–1728, April 1990.

[16] B. Moision and J. Hamkins, "Reduced Complexity Decoding of Coded Pulse-Position Modulation Using Partial Statistics," *The Interplanetary Network Progress Report*, vol. 42-161, Jet Propulsion Laboratory, Pasadena, California, pp. 1–20, May 15, 2005.
http://ipnpr/progress_report/42-161/161O.pdf

[17] J. Sun and O. Y. Takeshita, "Interleavers for Turbo Codes Using Permutation Polynomials over Integer Rings," *IEEE Transactions on Information Theory*, vol. 51, pp. 101–119, January 2005.

[18] O. Y. Takeshita, "On Maximum Contention-Free Interleavers and Permutation Polynomials over Integer Rings," *IEEE Transactions on Information Theory*, vol. 52, pp. 1249–1253, March 2006.

[19] S. Dolinar and D. Divsalar, "Weight Distributions for Turbo Codes Using Random and Nonrandom Permutations," *The Telecommunications and Data Acquisition Progress Report 42-122, April–June 1995*, Jet Propulsion Laboratory, Pasadena, California, pp. 56–65, August 15, 1995.
http://ipnpr/progress_report/42-122/122B.pdf

[20] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, Cambridge, Massachusetts: The M.I.T. Press, 1961.