Galois Field Hardware Architectures for Network Coding

Aishwarya Nagarajan
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Wisconsin, USA
anagarajan@wisc.edu

Michael J Schulte
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Wisconsin, USA
schulte@engr.wisc.edu

Parameswaran Ramanathan
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Wisconsin, USA
parmesh@engr.wisc.edu

***Abstract*- This paper presents and analyzes novel hardware designs for high-speed network coding. Our designs provide efficient methods to perform Galois field (GF) dot products and matrix inversions, which are important operations in network coding. Encoder designs that that perform GF dot products and vary with respect to the number of messages combined, Galois field size, and input message size are implemented and analyzed to evaluate design tradeoffs. We investigate single cycle, multi-cycle, and pipelined designs with and without feedback mechanisms for encoding multiple sets of messages. The decoder is implemented as a multi-cycle design and performs GF matrix inversion followed by multiple GF dot products. Our designs are synthesized with a 65nm standard cell library and compared in terms of area, clock period and throughput. Designs combining four messages achieve throughputs of more than 30 Gbps. Our designs can scale to achieve much higher throughput through the use of additional hardware.**

Keywords: Network coding, Content Distribution Networks, Galois field arithmetic, Matrix inversion, Gauss-Jordan elimination, Router designs

I. INTRODUCTION

About ten years ago, Ahlswede *et al.* published a landmark paper in which they proposed a novel technique called *network coding* [1]. In a conventional network, a router simply forwards its incoming packets on appropriate outgoing links. In contrast, in network coding, a router linearly combines a specified number of incoming packets before forwarding them on the appropriate outgoing links. The receivers of such network coded packets recover the original packets through a linear decoding operation on the appropriate set of the combined packets. Ahlswede *et al.* showed that, by using this technique, one can significantly increase the data rate achievable in a multicast application, where a single source node sends a stream of packets to more than one receiver.

Since the work by Ahlswede *et al.* [1], hundreds of papers have shown that the basic idea of network coding can be extended to enhance performance in both wired and wireless networks. For example, network coding can significantly reduce the delay of information exchange in peer to peer networks [15, 16]. Network coding can also improve the performance of unicast communication in wireless ad hoc networks [3] and can be used for available bandwidth estimation in overlay networks [17, 18].

Software implementations of network coding have been investigated for graphics processing units [8] and multi-core processors with vector extensions [11]. Although these implementations are flexible, since they are implemented using programmable processors, they require significant power and area, and do not achieve the Gigabit throughputs needed for high-speed routers.

Despite the tremendous promise of network coding, dedicated hardware implementations of network coding have not yet been investigated. Hardware implementations for network coding have the potential to allow network coding to occur at router line speeds, with much greater performance and energy efficiency than software implementations. The main bottlenecks in network coding are Galois field (GF) dot products during network encoding and decoding, and GF matrix inversion during network decoding. We focus on these two key operations to accelerate network coding.

This paper presents the design and evaluation of dedicated hardware architectures that perform GF dot products and GF matrix inversion for network coding. It includes a detailed analysis of tradeoffs in area and performance as the GF size, numbers of messages combined, and method for combining messages are varied. Our architectures have been synthesized using the TSMC 65nm standard cell library. Our synthesis results for a four message network coder indicate that throughputs of more than 30 Gbps are achievable with an area of roughly 2000 μm^2 for encoding, and 10,000 μm^2 for decoding. Thus, the proposed hardware designs have the potential to provide network coding capabilities in high-speed routers with low area overhead. Furthermore, by utilizing additional parallel hardware, our designs can scale to achieve much higher throughput. Novel contributions of this paper include:

- The first dedicated hardware architectures for network coding
- Detailed area-performance tradeoff analysis of GF dot product units
- Novel dedicated hardware architecture for GF matrix inversion, and Application and tradeoff analysis of the

GF dot product and matrix inversion units to high-speed network coding.

This paper is organized as follows. Section II provides background information and discusses related work on network coding and Galois field arithmetic. Section III presents our novel architectures for network encoding and decoding. Section IV gives results from our designs and analyzes design tradeoffs. Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. *Galois Field Arithmetic*

Galois Field (GF) arithmetic is a powerful algebraic tool, employed in many coding techniques, including network coding and Reed-Solomon coding. A brief explanation of key concepts and operations in GF arithmetic follows. Further details appear in [6].

A GF(2^m) field is an extension of the field GF(2), with elements {0, 1}. The operations defined in GF(2) are addition and multiplication, each performed modulo 2. When GF(2) is extended to GF(2^m), the result is a vector field of dimension *m* over GF(2). Elements of GF(2^m) can thus be represented as *m*-bit binary words. The field is characterized by the irreducible polynomial, *f(x),* where

$$f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1 x + f_0, \quad (1)$$

with f_i in GF(2). All 2^m elements in GF(2^m) can be represented by means of the vector basis {α^{m-1},….., α^1, α^0}, where α is a root of the irreducible polynomial *f(x)* and is called a primitive element of the field. This basis allows an element *A* in GF(2^m) to be expressed as

$$A = a_{m-1}\alpha^{m-1} + \ldots + a_1\alpha^1 + a_0\alpha^0, \quad (2)$$

with a_i in GF(2). Thus, elements of GF(2^m) can be associated with polynomials that have coefficients in GF(2), where the bit in the i^{th} position represents the coefficient of α^i. For example with GF(2^4), the element (1010) corresponds to $1\alpha^3 + 0\alpha^2 + 1\alpha^1 + 0\alpha^0$.

Addition and multiplication in GF(2^m) can be viewed as polynomial addition and multiplication. Since the polynomial's coefficients belong to GF(2), operations at the coefficient level are taken modulo-2. Thus, addition in GF(2^m) is performed by bitwise XORing the *m* coefficients of the two polynomials being added. Similarly, multiplication is computed as the modulo-2 sum of shifted partial products, where the sum is computed using bitwise XOR operations. The result of a GF(2^m) multiplication is a (*2m-1*)-bit word that represents a degree (*2m-2*) polynomial, in what is called extended form. To achieve closure, the extended form polynomial is reduced modulo *f(x).* This is equivalent to calculating the remainder from the extended form polynomial divided by *f(x).*

Early designs for GF multiplication used a serial approach [22]. Although serial GF multipliers have low hardware requirements, their delay increases linearly with the size of the input operands. Consequently, several parallel designs for GF multipliers have been developed, including Mastrovito multipliers [19, 20], Systolic GF multipliers [21], parallel array GF multipliers [5] and tree GF multipliers [4].

Gao and Parhi investigate VLSI designs for low latency and low power GF multipliers [5]. They present and discuss the logic structure, circuit design and physical mapping of GF multipliers. With their proposed architectures and physical mapping, an irregular balanced parallel tree GF multiplier can be implemented as easily as a regular GF multiplier. The custom VLSI implementations of these multipliers over GF(2^8) show that the irregular GF tree multiplier has 53% less delay and 58% less power consumption than a regular GF multiplier. Gao and Parhi also present an algorithm to parallelize the polynomial reduction of the extended form polynomial [5]. We employ their algorithm, called Parallel Polynomial Reduction (PPR), in our GF multiplier designs to reduce the extended form polynomial modulo *f(x).*

Garcia and Schulte present the design of a combined 16-bit binary and dual Galois Field multiplier, capable of performing either a 16-bit two's complement or unsigned multiplication, or two independent 8-bit GF(2^8) multiplications in SIMD fashion [4]. The combined multiplier is designed by modifying a conventional binary tree multiplier. It uses a novel wiring methodology to provide two simultaneous GF(2^8) multiplies with a minor impact on area and delay.

Csanky presents techniques for fast matrix inversions, but does not examine matrix inversion over Galois fields [12]. Litow and Davida introduce Boolean circuits that invert a single Galois field element [13].

As described in Section III, our designs use the Gauss-Jordan Elimination method [7] to perform GF matrix inversion and parallel techniques to perform GF dot products. They employ high-speed optimized modules for multiplication in GF(2^m) to implement GF matrix inversion and GF dot products.

B. *Network Coding*

In the discussion that follows, all the vector elements are indicated in bold text. Linear network coding using a chosen GF, GF(2^m), is implemented as follows. A message $\boldsymbol{b}_i$, $1 \le i \le n$ can be viewed as a vector of *k m*-bit blocks, b_{ij}, $1 \le j \le k$, where each b_{ij} belongs in GF(2^m). Given *n* input messages $\boldsymbol{b_1, b_2, \ldots, b_n}$ linearly combined to obtain an encoded message

$$\boldsymbol{e} = \sum_{j=1}^{n} c_j . \boldsymbol{b}_j \quad (3)$$

where each c_j belongs in GF(2^m), and where the addition and multiplication are performed over GF(2^m).

The decoding proceeds when the node receives n linearly independent encoded messages $[\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n}]^T$. An $n \times n$ coefficient matrix C is formed, using the coefficients included in each message. . The original messages $[\boldsymbol{b_1}, \boldsymbol{b_2}, \ldots, \boldsymbol{b_n}]^T$ are obtained as:

$$[\boldsymbol{b_1}, \boldsymbol{b_2}, \ldots, \boldsymbol{b_n}]^T = C^{-1}[\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n}]^T \quad (4)$$

Thus, computing the inverse of C is the first step to obtain $\boldsymbol{b_j}$. C is only invertible when it is full rank, i.e., when its rows are linearly independent. The next step is the GF matrix-vector multiplication of C^{-1} and $[\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_n}]^T$, which can be performed using n GF vector dot products.

Shojania and Li use off-the-shelf processors to accelerate network coding through multithreaded processing and SIMD vector instructions [11]. They later extend their work by implementing network coding with commodity off-the-shelf many-core GPUs, using a framework that they call Nuclei [8]. They show that a combined CPU-GPU encoding approach achieves encoding rates of up to 116 MB/second. Their design is oriented towards media streaming servers, in which hundreds of peers are served concurrently.

With respect to decoding, Shojania and Li recognize that owing to its higher computational complexity and limited opportunity for parallelism, the performance is poorer [8]. Decoding involves matrix inversions in the Galois field, and decoding a particular block cannot begin until the previous block is decoded, provided all the source blocks are linearly independent. They utilize a software-based Gauss-Jordan Elimination technique for matrix inversion that executes on the GPU. Their decoding occurs progressively, wherein each coded block along with its associated coefficients is decoded partially. They also consider the scenario where the n coded blocks are buffered and then decoded.

Unlike previous work on network coding, our approach uses dedicated hardware, rather than software running on CPUs or GPUs. Our designs are scalable with respect to size of the Galois field and the number of input messages used to form the final encoded message.

III. DESIGN

In the discussion to follow, n is the number of messages being encoded or decoded and m is the size of the Galois Field. Each message is assumed to be of size $L = k \cdot m$ bits for some positive integer k. The encoding output is a single $k \cdot m$-bit message, where each m-bit block of the output is a network encoding of the corresponding m-bit blocks of the n input messages. The decoding output is n $k \cdot m$-bit messages, where each m-bit block in an output message is the network decoding of the corresponding m-bit blocks of the n input messages. The overall Encoder and Decoder Architectures are illustrated in Figures 1 and 2, respectively. In the rest of this section, we focus on the hardware to compute one of the m-bit blocks in the output encoded message and the n m-bit blocks in the output decoded messages.

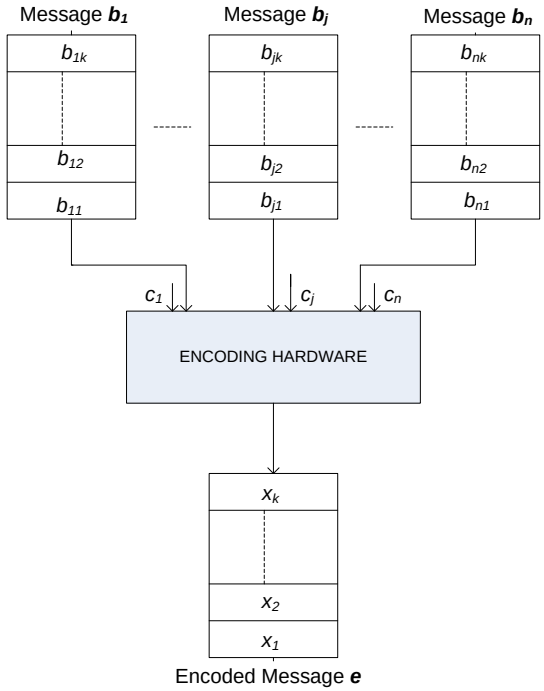

Figure 1: Overall Encoder Architecture

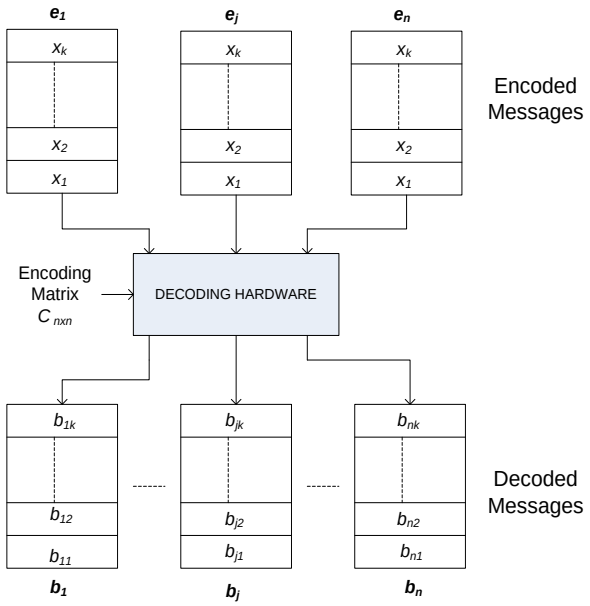

Figure 2: Overall Decoder Architecture

Our encoder and decoder architectures are designed primarily for network routers, but they should also be useful in performing high-speed network coding in other types of systems, such as media servers.

A. Encoder Architecture

Figure 3 shows a block diagram of our network encoder, which computes GF dot products, as specified in Equation (3). A single GF dot product unit takes one *m*-bit block from each message, multiplies them with corresponding GF coefficient and then sums them using a XOR tree. The output is one block of the encoded message. If more than *n* messages need to be combined, the network encoder can be expanded to include additional GF multiply-reduce units and XOR operations. Alternatively, a feedback register can be added and multiple passes through the network encoder can be used to compute a larger dot product. With the feedback approach, computing the dot product of *n* pairs of vector elements with *p* multiply-reduce units and a (*p+1*)-input XOR tree takes *n/p* iterations. The optional feedback register and feedback path are indicated with dashed lines in Figure 1. To encode one *k·m*-bit message, *k* dot products are performed over GF(2^m). To improve the throughput of the encoder, multiple dot product units can be used to compute multiple message blocks in parallel.

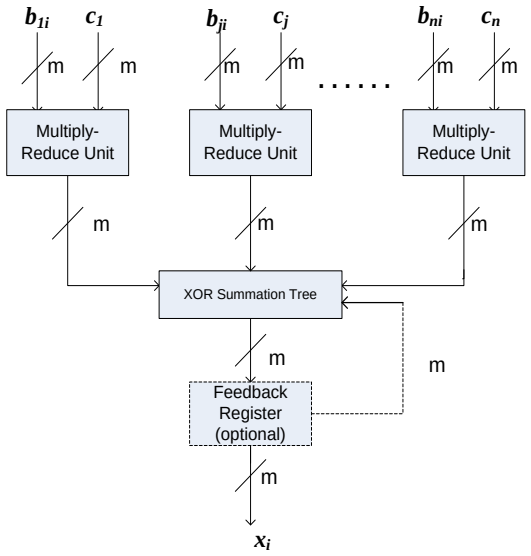

Figure 3: Block Diagram of the Galois Field Dot Product Unit

Each GF(2^m) multiplier takes in an *m*-bit block b_{ji} from a message queue, multiplies it with an arbitrary *m*-bit coefficient c_j and produces a (2*m* - 1)-bit extended form polynomial. Our GF(2^m) multiplier utilizes the equivalent of m^2 parallel AND gates to generate *m m*-bit partial products, followed by a tree of (m^2 - 2*m* + 1) XOR gates to sum the partial products using GF addition.

After GF multiplication, the PPR technique described below is used to reduce the (2*m* - 1)-bit extended form polynomial and produce a final *m*-bit product. In PPR [5], the extended polynomial *P(α)* can be separated into two parts, such that:

$$P(\alpha) = \sum_{i=m}^{2m-2} p_i \alpha^i + \sum_{i=0}^{m-1} p_i \alpha^i, \quad (5)$$

PPR uses Equation (2) and the following identity

$$\alpha^i = A_i(\alpha) = \sum_{j=0}^{m-1} a_{i,j} \alpha^j, m \le i \le 2m-2 \quad (6)$$

where $A_i(\alpha)$ is the canonical representation of the sum of the field elements α^i that appear in the first summation in Equation (5). $A_i(\alpha)$ can be added with the second summation in Equation (5), which represents one field element in canonical form, to produce the final result. As an example with *m = 4*, suppose *f(x)* = x^4+x+1 (10011) and the extended form polynomial is *P(α)* = $x^6+x^5+x^2+1$ = (1100101). For this *f(x)*, α^4 = (0011); α^5 = (0110); α^6 = (1100). The final result after modulo reduction, for *P(α)* = (1100101), is computed using GF addition as

$$1\alpha^6 + 1\alpha^5 + 0\alpha^4 + (0101) = (1111)$$

Since the irreducible polynomial is known, our design (illustrated in Figure 4) utilizes pre-computed values for α^i, for $m \le i \le 2m$-2, and performs the polynomial reduction in a tree with logarithmic delay using the technique by Gao and Parhi [5]. The reduction unit requires (*m*-1)·*m* AND gates to determine the elements of the first summation and (*m*-1)·*m*-bit XOR gates to compute the reduced polynomial using a tree structure using the following equation.

$$P(\alpha) = \sum_{i=m}^{2m-2} p_i \alpha^i + \sum_{i=0}^{m-1} p_i \alpha^i$$

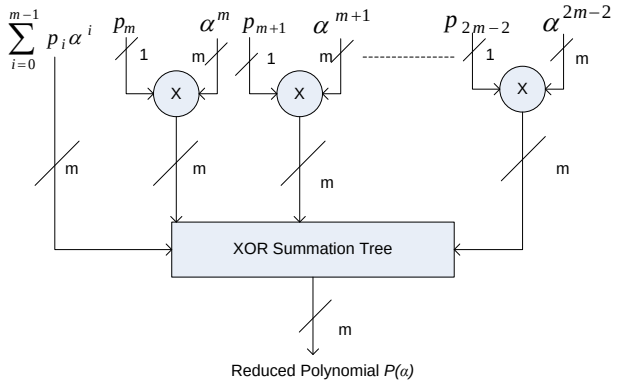

Figure 4: Block Diagram of the Reduction Unit

In Figure 3, the dot product unit first performs *n* multiply-reduce operations in GF(2^m) and then XOR sums the reduced products to produce x_j. An alternative approach, which produces the same results, is to first perform *n* multiply operations without reduction in GF(2^m), then XOR sum the *n* extended form polynomials,

and finally perform one polynomial reduction of the result to produce x_j. Although the second approach requires n x (2m-1)-bit extended products to be added in the XOR tree, it only requires a single reduction unit. We evaluate both types of GF dot product designs in Section IV.

B. Decoder Architecture

Network decoding, which involves GF matrix inversion followed by GF matrix-vector multiplication to implement Equation (4), has higher computational complexity than network encoding. The matrix-vector multiplication of an n x n matrix with an n-element vector requires n dot product operations. Once the matrix inversion has been performed, the inverse matrix can then be used to decode blocks from all messages that were encoded using those coefficients. This is depicted in Figure 5a. In real systems, the GF matrix inversion can occur more slowly than the GF dot product, since the results of one matrix inversion is used to decode n $k \cdot m$-bit messages, which requires $n \cdot k$ dot product operations.

Our network decoder design utilizes Gauss-Jordan Elimination to perform matrix inversion with matrix elements in GF(2^m). Gauss Jordan Elimination [6, 7] is a widely used technique for finding the inverse of a square matrix. The square matrix is augmented with the identity matrix of the same dimensions. Elementary row operations are then performed on the augmented matrix until it reaches reduced row echelon form, from which the inverse matrix can be obtained, as per the following equation:

$$[CI] => C^{-1}[CI] => [IC^{-1}] \ , \qquad (7)$$

where C is the original matrix, C^{-1} its inverse, and I is the identity matrix. Elementary row operations include combinations of pivoting, multiplication, and addition on the rows and columns of a matrix, such that they change the matrix image but not its kernel, i.e, the solution set of the system of linear equations represented by the matrix. A matrix is in reduced row echelon form if

- All nonzero rows (rows with at least one nonzero element) are above any rows of all zeros.
- The pivoting element (the first nonzero number from the left) of a nonzero row is always strictly to the right of the leading coefficient of the row above it.
- Every leading coefficient is 1 and is the only nonzero entry in its column.

Further details are found in [7].

The matrix inversion module, shown in Figure 5b, takes in the matrix of coefficients used in the encoding process, C, and obtains C^{-1} in GF(2^m) using Gauss-Jordan elimination. The register file stores the augmented matrix elements. Each cycle, one matrix element is supplied as an input to a multiply-reduce unit. The other input to it can either be a value from the register file, or the multiplicative inverse value of an element from the register file. This is determined by the control FSM using the signal *inverse_sel = ((cycle % n) == 0)*, where *cycle* is the cycle count starting from 0. The control FSM also determines which inputs from the register file must be used in a cycle by computing the appropriate indexes. Once the multiply-reduce units computes their results, these are supplied to the XOR gates. The second inputs to the XOR gates are also supplied from the register file, which also stores the result of the XOR.

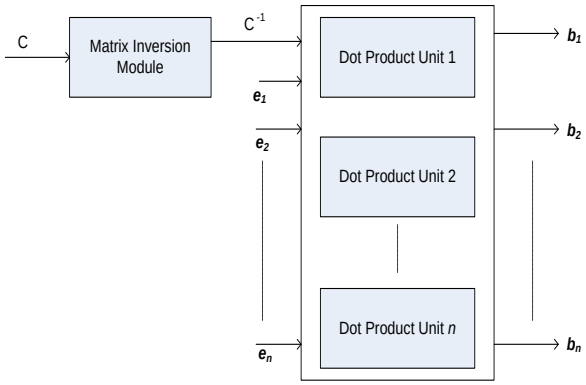

Figure 5a: Block Diagram of the Decoder Unit

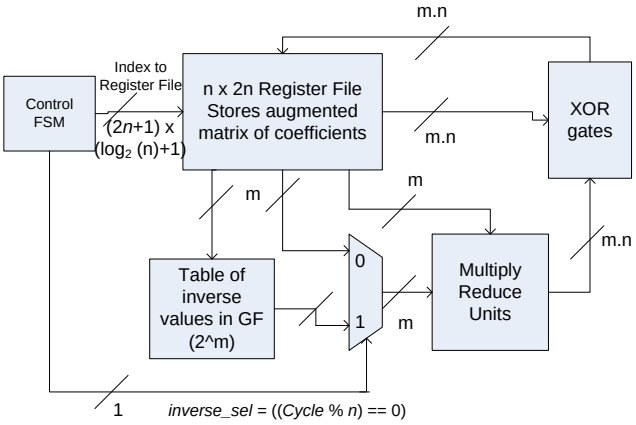

Figure 5b: Block Diagram of the Matrix Inversion Unit

We have designed an n x n matrix inverter that takes n^2 clock cycles to invert the matrix. It uses n multiply-reduce units and $n \cdot m$ XOR gates. In clock cycle 0, the matrix inverter determines the GF multiplicative inverse of the pivoting element of row 1 (typically the diagonal element) via a table lookup. It then uses the n multiply-reduce units described earlier to multiply the elements of row 1 with the inverse of the pivot element, and obtain a new pivot row with a 1 in the pivot position. In the subsequent (n - 1) clock cycles, it processes the (n - 1) other rows to get a 0 in the positions above and below the pivot positions. This is accomplished by multiplying the element above/below the pivot element with each element of the pivot row and then XORing each of these with the corresponding element of the row being processed. Figure

6 shows the multiply-reduce units and XOR gates that process each row. Based on the *inverse_sel* signal, either the inverse of the pivot element or a value from the register file is supplied to the multiply reduce unit. The other input to the multiply-reduce units is also supplied by the register file. The output of this is supplied to the XOR gates. The other input to the XOR gates also comes from the register file, and the result is written back to it. This procedure is repeated every n clock cycles with the next pivot row and pivot position. During these clock cycles, the control FSM checks if the pivot element is a 0, which indicates that a new pivot element must be selected. In such a case, the pivot elements of the subsequent rows are checked in parallel and the first row found with a non-zero pivot element is swapped with this row. If no such row can be found, the matrix is deemed non-invertible and a flag is raised.

We have designed a matrix inverter that decodes n = 4 source messages encoding in GF(2^8), i.e., m = 8. We employ four multiply-reduce modules and 32 2-input XOR gates in this module. We can modify the baseline design for smaller area but with a larger latency by reusing the same multiply-reduce units over more clock cycles and processing different portions of a row in each clock cycle. In order to use p multiply-reduce units and $p \cdot m$ XOR gates to process a n x n matrix, where $n > p$, the design can process p elements of a row in each clock cycle, and require n/p clock cycles to process a row, and n^3/p clock cycles to invert the entire matrix. Alternatively, a larger GF matrix inversion unit that takes fewer clock cycles than the baseline to invert the matrix can be employed. This approach utilizes p multiply-reduce units and $p \cdot m$ XOR gates to process a n x n matrix, where $p > n$. It takes 1 clock cycle to process p/n rows and n^3/p clock cycles to invert the entire matrix.

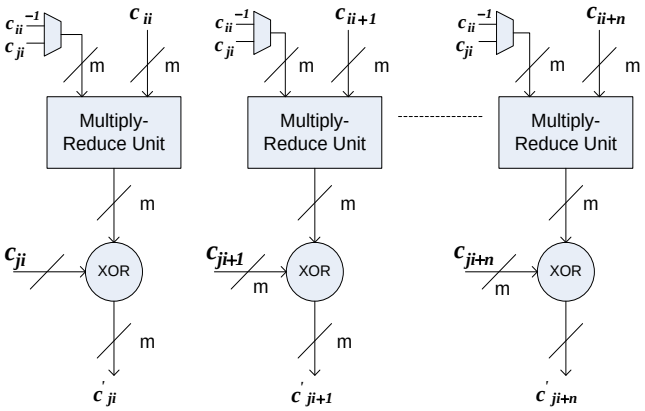

Fig. 6: Structure of the Main Datapath of the Matrix Inversion Module

Once the inverted coefficient matrix, C^{-1}, is obtained, the design decodes the source messages by multiplying the inverse matrix and the encoded messages over the chosen irreducible polynomial, as per Equation (4), with n GF dot product operations. These GF dot products can be computed using the same hardware that is used for encoding and is shown in Figure 3.

IV. RESULTS AND ANALYSIS

To evaluate our network coding architectures, we developed parameterized Verilog modules for GF dot products and matrix inversion units. Our designs were synthesized using Synopsys Design Compiler and the TMSC 65nm standard cell library for an operating voltage of 1.00 Volt and a temperature of 25 degrees Celsius. Input and Output delays were set to 5% of the clock period. We optimize our designs for delay and direct the synthesis tool to flatten (i.e., the designs are ungrouped and the design hierarchy is remove to improve area and delay). We present results for area (in μm^2), clock period (in ns) and throughput (in gigabits per second). The throughput for the encoders is calculated as the number of bits output per second, which is calculates as *throughput = m/critical path delay*.

A. Galois Field Dot Product Units

For our baseline designs, we considered hardware architectures for a network router that encodes and decodes up to four messages, with arbitrary coefficients and a fixed, irreducible polynomial in GF(2^8). Thus, in our baseline design $n = 4$ and $m = 8$. Figures 7 through 9 show the area, clock period and throughput respectively for our baseline GF dot product unit. These figures use the abbreviations SC for Single Cycle, PL for Pipelined, RU for Reduction Unit, and OR for *optimize_registers* performed in synthesis. The *optimize_registers* command in Synopsys Design Compiler attempts to balance the delay between pipeline registers at the expense of additional area.

Our pipelined designs take three cycles to produce the first result. In GF dot product units with multiple GF reduction units, the pipeline registers are placed between the multiplication and reduction stages, and between the reduction stage and the XOR summation tree. In GF dot product units with a single GF reduction, they are placed between the multiplication stage and the XOR summation tree, and between the XOR summation tree and the reduction stage.

As shown in Figures 7 to 9, pipelined designs have a shorter clock period and higher throughput than non-pipelined designs, but have much higher area due to pipeline registers. Utilizing the *optimize_registers* option further improves the clock period and throughput, but increases area compared to pipelined designs that do not use this option. With our baseline design, a pipelined GF dot product unit that encodes four messages and utilizes *optimize_registers* has an area of roughly 2000 μm^2, a critical path delay of 0.26 ns, and a throughput of over 30 Gbps. In comparison, a 32-bit by 32-bit parallel multiplier

implemented in the same technology and under the same operating conditions has an area of roughly 6788μm^2 and a critical path delay of 0.45 ns. The designs with a single reduction unit achieve similar or better critical path delays than their counterpart designs with four reduction units and also require less areas. For pipelined designs, encoders synthesized with *optimize_registers* achieve much better throughputs than designs synthesized without *optimized_registers* (30.77Gbps vs. 26.67 Gbps). Hence, for the subsequent results, we synthesize our designs with a single reduction unit, and the pipelined designs use the *optimize_registers* feature.

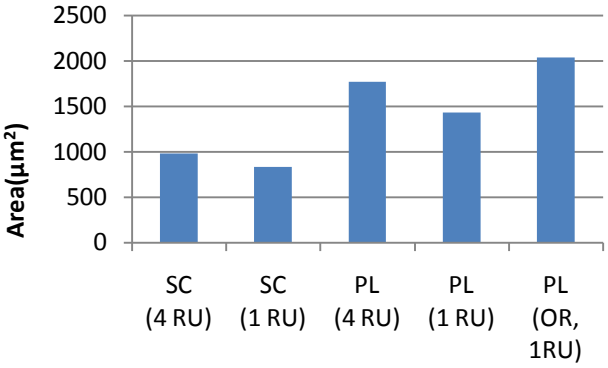

Fig. 7: Comparison of areas for our baseline GF dot product unit

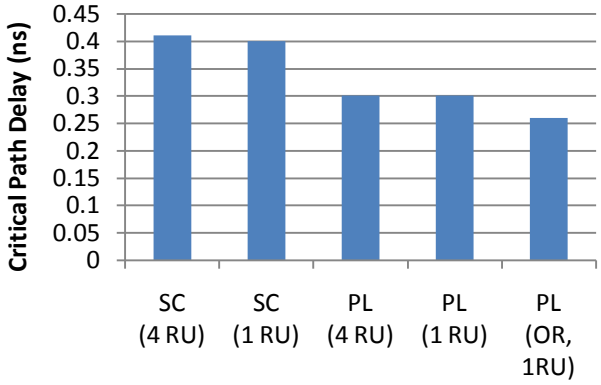

Fig. 8: Comparison of critical path delays for our baseline GF dot product unit

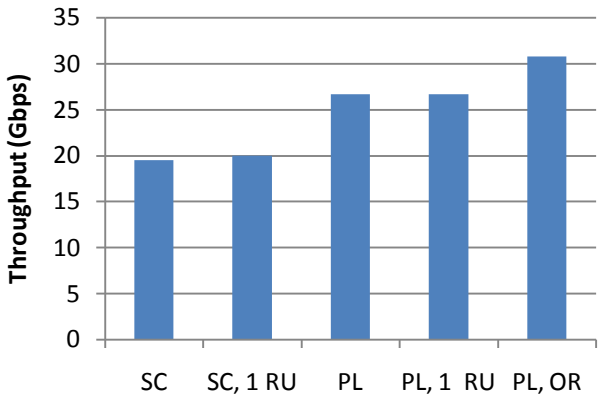

Fig. 9: Comparison of throughputs for our baseline GF dot product unit

We also investigated the impact of varying the Galois field size, *m*, on area, delay, and throughput, while keeping the number of messages combined the same, i.e., $n = 4$. As discussed earlier, in these designs, a single reduction unit is used and *optimize-registers* is enabled for pipelined designs. As shown in Figure 10, the area of the GF dot product units increases quadratically as m increases. . Figure 11 shows that for Single Cycle designs, the critical path delay increases by a factor of around 1.5 when *m* is doubled. For pipelined designs, doubling *m* has little impact on the critical path delay. This is because for values of *m* from 2 to 8 there are only a few gates on the critical delay path. As shown in Figure 12, when *m* is incremented by 1, throughputs increase by roughly a factor of 1.2 for Single Cycle designs and by roughly a factor of 2 for Pipelined designs. However, despite having smaller area and critical path delays, smaller fields make it more probable that the set of coefficients corresponding to the set of received encoded messages are linearly dependent, preventing recovery of the original messages [23].

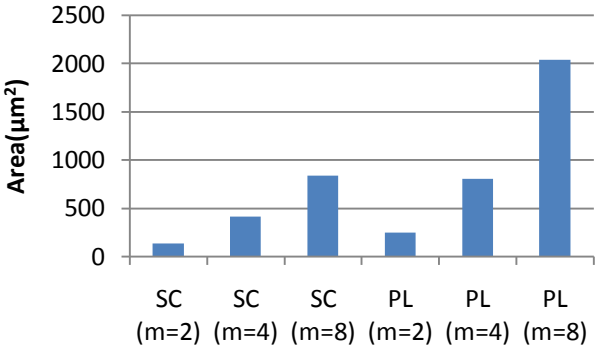

Fig. 10: Areas for GF dot product units with varying *m*

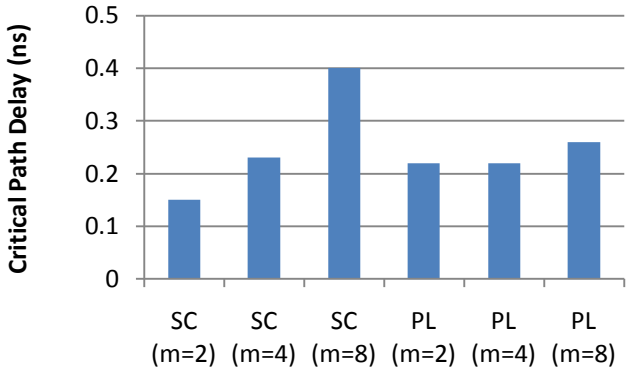

Fig. 11: Critical path delays for GF dot product units with varying *m*

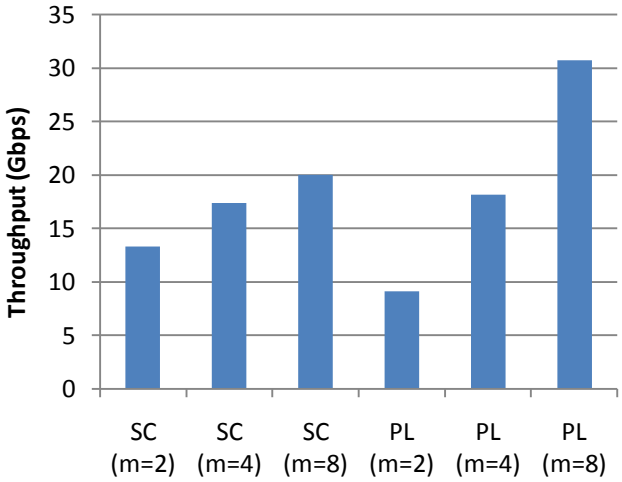

Fig. 12: Throughputs for GF dot product units with varying *m*

To investigate the scalability of our GF dot product units, we determined the impact on area, delay, and throughput of varying the number of messages combined, *n*, while keeping the size of the Galois Field the same, i.e., $m = 8$. We have implemented designs with $n = 2, 4, 8$ and 16. As discussed earlier, in these designs, a single reduction unit is used and *optimize-registers* is enabled for pipelined designs. Figure 13 shows that area increases linearly with *n* for Single Cycle designs and Pipelined designs. As shown in Figure 14, the critical path delays for both Single cycle and Pipelined designs increases logarithmically with *n*. As can be seen in Figure 15, throughputs decrease as *n* increases, due to the increase in the critical path delay. The Pipelined designs have higher throughputs than the corresponding Single Cycle designs. However, as *n* increases, more messages can be combined, which can be used to improve network bandwidth.

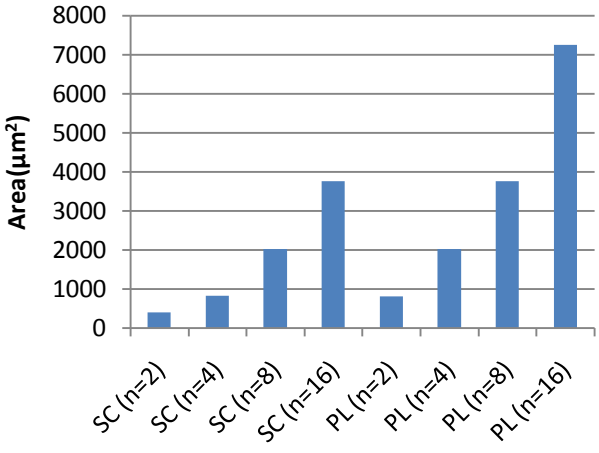

Fig. 13: Areas for GF dot product units with varying *n*

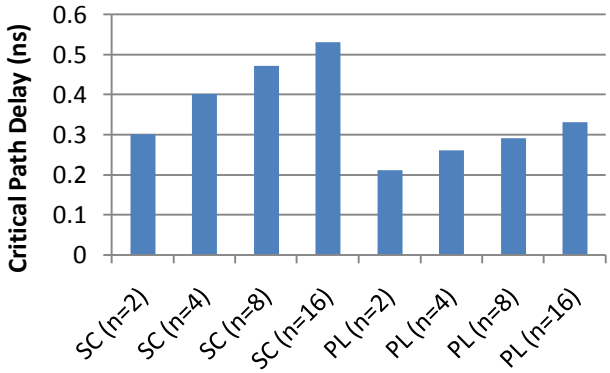

Fig. 14: Critical path delays for GF dot product units with varying *n*

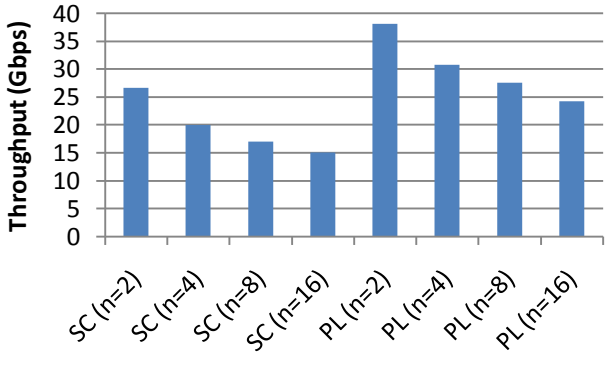

Fig. 15: Throughputs for GF dot product units with varying *n*

We also compared the area and throughput of pipelined and single-cycle designs with and without feedback. As discussed in Section III.B, designs with feedback designs utilize a feedback register, *p* GF multipliers, a $(p+1)$-input XOR summation tree, and n/p passes through the dot product unit to combine *n* symbols from *n* messages (one symbol per message).

The results for single cycle and pipelined designs are summarized in Figures 16 and 17, respectively, where we plot the throughput of the designs with respect to the their area for $m = 8$ and varying values of n and p. The top left portion of the chart corresponds to high throughput and low area.

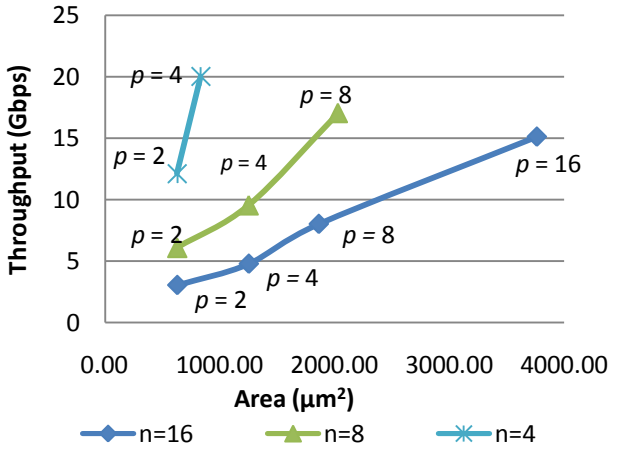

Fig. 16: Throughput vs. area for single cycle designs with and without feedback

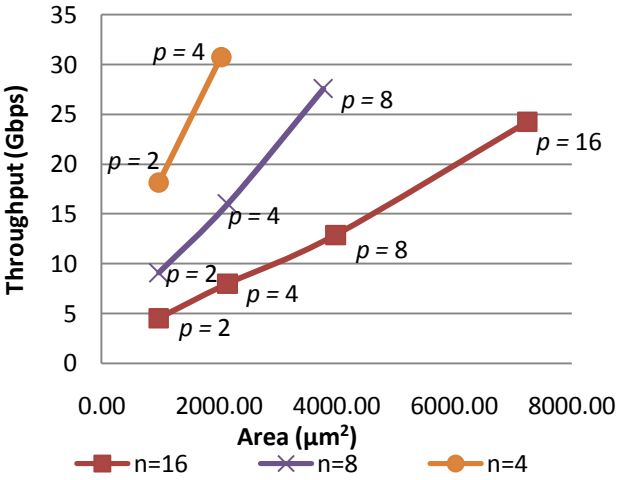

Fig. 17: Throughput vs area for pipelined designs with and without feedback

These plots indicate that, in general, Pipelined designs achieve better throughputs than their counterpart Single cycled designs at the cost of larger area. For a given value of n, increasing p increases both throughput and area. Lines with steep slopes indicate situations in which a relatively small increase in area yields a large increase in throughput. Since smaller values of n have steeper lines, this indicates that they achieve a larger increase in throughput for a given amount of area, than designs with larger values of n.

B. Galois Field Matrix Inversion Units

Our GF matrix inversion units use multiple cycles to perform matrix inversion with less area than a fully parallel implementation. For our baseline design with $m = 8$ *and* $n = 4$, the GF matrix inversion unit uses four multiply-reduce units and 32 XOR gates over 16 clock cycles to generate the inverted matrix, including performing checks for linear independence. The GF matrix inversion unit operates at a clock speed of 0.45ns and occupies an area of 10,540 cell units. It takes 7.2ns to produce a valid result for a 4 x 4 matrix. The larger area for this unit is due to the presence of a 32-entry by 8-bit register file for storing the augmented matrix of coefficients, as well as the state registers for a 16-state FSM used for the inversion.

To obtain the source messages following the matrix inversion, the GF dot product units are utilizes repeatedly to perform GF matrix-vector multiplication.

V. CONCLUSIONS

High-speed network coding has the potential to provide significant benefits to future networking systems. We have presented and analyzed architectures for network coding that implement the important operations of GF dot products and GF matrix inversion. We have also investigated tradeoffs due to variations in the encoder design. To achieve faster encoding and decoding, multiple modules may operate in parallel, which results in an increase in area. To obtain smaller designs, we can reuse modules over multiple clock cycles. The feedback mechanism for GF dot products in particularly useful in that it provides the ability to vary the number of messages encoded at a fixed hardware cost.

Considering the minor area overheads and the comparable throughputs achieved, it would be beneficial to incorporate these designs in routers, content distribution servers and other applications to provide the bandwidth and transmission efficiency of network coding.

REFERENCES

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yueng, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, July 2000.

[2] C.Fragouli, J. L. Boudec, and J. Widmer, "Network coding: an instant primer," SIGCOMM Computer Communications Review, vol. 36, no. 1, pp. 63–68, 2006.

[3] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in the air: practical wireless network coding," in Proc of the ACM SIGCOMM, pp. 243–254, September 2006.

[4] J. Garcia and M. J. Schulte, "A Combined 16-bit Binary and Dual Galois Field Multiplier," in Proc. of

IEEE Workshop on Signal Processing Systems, pp. 63-68, 2002.

[5] L. Gao and K. K. Parhi, "Custom VLSI Design of Efficient Low Latency and Low Power Finite Field Multiplier for Reed-Solomon Codec," in Proc. of *IEEE International Symposium on Circuits and Systems,* pp. IV 574-577, 2001.

[6] R. Lidl and H. Niederreiter, "Introduction to Finite fields and their applications," *Cambridge University Press*, 1994.

[7] K. A. Atkinson, "An Introduction to Numerical Analysis, (2nd ed.)," *John Wiley & Sons, New York,* 1989.

[8] H. Shojania, B. Li, X. Wang, "Nuclei: GPU-accelerated many-core network coding," in Proc. of *IEEE INFOCOM*, pp. 459-467, April 2009.

[9] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of Allerton Conference on Comm., Control, and Computing,* October 2003.

[10] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. of IEEE INFOCOM,* March 2005.

[11] H. Shojania and B. Li, "Parallelized Network Coding With Hardware Acceleration," in *Proc. of the 15th IEEE International Workshop on Quality of Service (IWQoS),* June, 2007.

[12] L. Csanky, "Fast Parallel Matrix Inversion Algorithms," *SIAM J. Comput. Volume* 5, Issue 4, pp. 618-623, 1976.

[13] B. E. Litow and G. I. Davida, "O(log(n)) Parallel Time Finite Field Inversion," *Lecture Notes in Computer Sciences,* vol. 319, pp. 74-80, 1988.

[14] Texas Instruments, TMS320C64x Technical Overview.

[15] M. Yang and Y. Yang, "Peer-to-peer File Sharing based on Network Coding," in Proc. of *28th International Conference on Distributed Computing Systems,* pp. 168–175, June 2008.

[16] M. Yang and Y. Yang, "Topology affects the efficiency of network coding in peer- to-peer networks," in Proc. of *IEEE International Conference on Communications*, pp. 5591–5597, May 2008.

[17] Y. Cui, Y. Xue, and K. Nahrstedt, "Max-min overlay multicast: Rate allocation and tree construction," in Proc. of *IEEE International Workshop on Quality of Service*, pp. 221–231, June 2004.

[18] N. Sundaram, "Distributed Multirate Streaming In Overlay Networks," PhD Dissertation, University of Wisconsin-Madison, 2008.

[19] E.D. Mastrovito, "VLSI Designs for Multiplications over Finite Fields GF(2^m)," in Proc. of *sixth International Conference on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-6)*, pp. 297-309, 1988.

[20] T. Zhang and K. K. Parhi, "Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible Polynomials," *IEEE Transactions on Computers*, vol. 50, pp. 734-749, 2001.

[21] C. Yeh, I. S. Reed, and T. K. Trouong, "Systolic Multipliers for Finite Field GF(2^m)," *IEEE Transactions on Computers*, vol. C-33, pp. 357, 1984.

[22] B. A. Laws and C. K. Rushforth, "A Cellular-Array multiplier for GF (2^m)," *IEEE Transactions on Computers,* vol. C-20, pp. 1573-1578, Dec. 1971.

[23] I. S. Reed and G. Solomon, "Polynomial codes over certain Finite Fields, " *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, No. 2, pp. 300-304, June 1960.