SYSTOLIC INTERPOLATION ARCHITECTURES FOR SOFT-DECODING REED-SOLOMON CODES

Arshad Ahmed, Naresh R. Shanbhag, and Ralf Koetter

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.

ABSTRACT

We present a systolic algorithm for performing interpolation, a computationally intensive kernel found in algebraic soft-decoding of Reed-Solomon codes. We reformulate the interpolation algorithm, resulting in the *systolic interpolation algorithm*, which can compute a reduced number of candidate polynomial coefficients. Using the dependence graph of the algorithm, we realize a low-latency interpolation architecture and a high-throughput interpolation architecture. These architectures are compared against previously proposed architectures for a RS soft-decoder. We derive expressions for the latency of the systolic implementations and show that, for a reasonable hardware constraint, the low-latency systolic implementation reduces latency by 34% for a $[255, 239]$ RS code. For the same code and hardware constraints, the high-throughput implementation with a block pipelining depth of 5, increases throughput by 68%. In addition, the critical path of both the low-latency and the high-throughput implementation is smaller than that of previously proposed architectures.

1. INTRODUCTION

List-decoding algorithms [1], [2] and their soft-decoding [3] counterparts extend the coding gains obtained from Bounded Minimum Distance (**BMD**) decoding [4] of Reed-Solomon codes. These gains, obtained by decoding beyond half the minimum distance of the code, come at the price of increased complexity compared to BMD decoding. Algorithm level techniques for decreasing the complexity of list-decoding and soft-decoding have been proposed in [5], [6] and some preliminary architectural issues are dealt with in [6], [7]. The two main computational kernels in list-decoding are the bivariate polynomial based interpolation step which is followed by a bivariate polynomial factorization step. As was shown in [7], interpolation dominates over factorization both in terms of decoding delay and decoding resources. In this paper, we present systolic architectures for interpolation, derived by first reformulating the conventional interpolation algorithm and then folding the resulting regular iterative systolic dataflow graph.

Systolic architectures [8], [9] use an array of processing elements that periodically compute and forward data to neighboring elements. These architectures possess features such as modularity and regularity which result in efficient VLSI implementations. Systolic implementations of BMD decoding [10], [11], which are based either on the Berlekamp-Massey algorithm or the Euclid's algorithm are among the most compact, high-speed implementations reported in literature. This paper attempts to incorporate these desirable traits into RS soft-decoding implementations. Further, in the reformulated systolic interpolation algorithm, we can compute a reduced number of candidate polynomial coefficients. We utilize this feature to match the output of the interpolation kernel to the requirements of the factorization kernel, thus reducing interpolation computations and partially counteracting the increase in storage associated with a systolic implementation.

The rest of the paper is organized as follows. Section 2 is an overview of soft-decoding and the interpolation procedure. The reformulation leading to the systolic interpolation algorithm is explained in Section 3. The dependence graph of the systolic interpolation algorithm and architectures for systolic interpolation are presented in Section 4. Section 5 compares systolic architectures optimized for low-latency and high-throughput with previously reported architectures.

This work was funded by NSF grant: CCR 00-73490

2. OVERVIEW OF SOFT-DECODING AND INTERPOLATION

In this section, we will outline the main ideas of soft-decoding and the steps involved in the interpolation algorithm. A block diagram of the process of soft-decoding is shown in Fig. 1. The reader is referred to [3] for more details regarding the Koetter-Vardy front end. The Guruswami-Sudan list decoding process using bivariate polynomials $Q(X, Y)$ is explained in [1], [2]. We now lay out the basic terminology required to understand the interpolation process.

For an $[n, k]$ RS code, the input to the list-decoder is a set of s ordered triples of the form (x_i, y_i, m_i) where $0 \leq i < s$. The ordered pair (x_i, y_i), termed an *interpolation point* is a set of finite field elements in some $GF(2^q)$. The integer m_i is termed the required *multiplicity* at the interpolation point (x_i, y_i). A bivariate polynomial $Q(X, Y)$ is said to pass through the point (x_i, y_i) with multiplicity m_i if in $Q(X+x_i, Y+y_i)$, the coefficients of all terms $X^{\alpha}Y^{\beta}$ with $\alpha + \beta < m_i$ is zero. The two main steps involved in the Guruswami-Sudan list decoding process are as follows:

- **Interpolation step:** For an $[n, k]$ RS-code, given the set of non-trivial interpolation points (x_i, y_i) with $0 \leq i < s$ with corresponding multiplicities m_i, compute the *minimal polynomial* $Q(X, Y)$, which is the non-trivial polynomial of minimum $(1, k-1)$-weighted degree that passes through each of the interpolation points (x_i, y_i) with multiplicity m_i.
- **Factorization step:** Determine all the factors of the bivariate minimal polynomial $Q(X, Y)$, which are of the type $Y - f(X)$ with $\deg(f(X)) < k$. Each of the univariate polynomials $f(X)$ corresponds to a valid data polynomial.

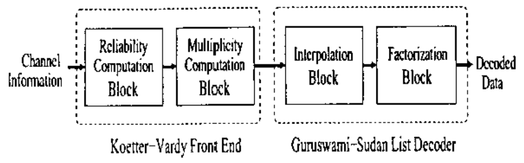

Fig. 1. Block diagram of the soft-decision decoder.

In this paper, we will confine ourselves to the interpolation step as applied to soft-decoding. The Fundamental Iterative Algorithm (**FIA**) first presented in [12] is used to determine the minimal polynomial. The pseudo-code of the interpolation algorithm for list-decoding based on the **FIA** is shown in Fig. 2. It is termed the Constraint Serial Interpolation Algorithm (**CSIA**), as it performs serially for each constraint, the operation of Discrepancy Coefficient Computation (**DCC**) followed by Polynomial Update (**PU**).

Each interpolation point (x_i, y_i) with multiplicity m_i imposes $\binom{m_i+1}{2}$ constraints, where each constraint corresponds to a unique (α, β) which satisfies $\alpha + \beta < m_i$. For each of these constraints, the coefficient of $X^\alpha Y^\beta$ in $Q_l(X + x_i, Y + y_i)$ for each of the candidate polynomials is determined in the **DCC** step. The coefficients $d^{(l)}_{\alpha\beta}$, termed *discrepancy coefficients*, are analogous to the discrepancies computed in the Berlekamp-Massey algorithm.

If a candidate polynomial has a non-zero discrepancy value then it needs to be modified to satisfy the current constraint. This is done in the **PU** steps. Step **PU0** determines the candidate polynomial of minimum $(1, k-1)$ weighted degree with a non-zero discrepancy and denotes its index *minl*. The step **PU1** is analogous to the Berlekamp-Massey update equations, with $Q_{minl}(X, Y)$ playing the role of the scratch polynomial. In Step **PU2**, the polynomial with index *minl* is multiplied with $(X + x_i)$. This forces the polynomial to satisfy the current constraint but also results in a unit increase in its weighted degree.

From a hardware implementation perspective, the following facts about the **CSIA** are noteworthy:

1. At any given instant, either the **DCC** or the **PU** operation is being performed, which lowers hardware utilization.
2. Increase in the size of the candidate polynomials (in Step **PU2**) increases complexity of both **DCC** and **PU** steps.
3. As the data structures and operations increase in size and complexity with iterations, mapping the algorithm to hardware is essentially a worst case mapping.

A technique to partially undo this serial nature of interpolation, with the intention of decreasing computational complexity was presented in [7], wherein all the discrepancy coefficients corresponding to a particular interpolation point were computed in parallel. In the next section, we elaborate on the idea behind this technique and extend it to realize a systolic algorithm.

3. REFORMULATED INTERPOLATION ALGORITHM

In Section 3.1, we present the reformulation of the interpolation algorithm. In Section 3.2, we re-index the operations in order to illustrate the systolic nature of the reformulated algorithm. We then show that it is possible to reduce the number of candidate polynomial coefficients computed in the reformulated algorithm.

```
Initialization:
$Q_0(X,Y) = 1, Q_1(X,Y) = Y, \cdots ; Q_t(X,Y) = Y^t$
$Wdeg_0 = 0, Wdeg_1 = k-1, \ldots ; Wdeg_t = t(k-1)$
Input: $(x_i, y_i, m_i)$; $0 \leq i < s$
  for $i = 0$ to $s-1$ do
    for $\alpha = 0$ to $m_i - 1$ and $\beta = 0$ to $m_i - 1 - \alpha$ do
      Discrepancy Coefficient Computation (DCC):
      $d^{(l)}_{\alpha\beta} = \mathrm{Coeff}(Q_l(X + x_i, Y + y_i), X^\alpha Y^\beta)$
                  $(l = 0, 1, \cdots, t)$
      Polynomial Update (PU):
      PU0: $minl = \mathrm{argmin}(Wdeg_k | d^{(k)}_{\alpha\beta} \neq 0, k \leq t)$
      PU1: $(0 \leq l \leq t$ and $l \neq minl)$
            $Q_l(X,Y) \Leftarrow d^{(minl)}_{\alpha\beta} \times Q_l(X,Y) +$
            $d^{(l)}_{\alpha\beta} \times Q_{minl}(X,Y)$
      PU2: $(l = minl)$
            $Q_{minl}(X,Y) \Leftarrow Q_{minl}(X,Y) \times (X + x_i)$
            $Wdeg_{minl} \Leftarrow Wdeg_{minl} + 1$
    endfor
  endfor
Output: $Q_{minl}(X,Y)$; $minl = \mathrm{argmin}(Wdeg_k; 0 \leq k \leq t)$
```

Fig. 2. The Constraint Serial Interpolation Algorithm (CSIA)

3.1. Principal idea driving reformulation

Let $d^{(l)}_{\alpha\beta}$ with $0 \leq l \leq t$ be the set of discrepancy coefficients computed at a constraint corresponding to a particular interpolation point (x_i, y_i) with $\alpha + \beta < m_i$. As in the pseudo-code of Fig. 2, the **PU** equations are given by:

$$\tilde{Q}_l(X,Y) = \begin{cases} d^{(minl)}_{\alpha\beta} \times Q_l(X,Y) & \\ +d^{(l)}_{\alpha\beta} \times Q_{minl}(X,Y) & \text{if } l \neq minl \\ Q_{minl}(X,Y) \times (X + x_i) & \text{if } l = minl \end{cases} \quad (1)$$

After performing the **PU** operations, the discrepancy coefficients $d^{(l)}_{\alpha'\beta'}$ corresponding to the next constraint at the interpolation point (x_i, y_i) can be determined using the updated candidate polynomials $\tilde{Q}_l(X,Y)$. Alternatively, we can first compute tentative discrepancy coefficients $\tilde{d}^{(l)}_{\alpha'\beta'}$, using the original candidate polynomials $Q_l(X,Y)$, (from RHS of (1)) and then update the tentative discrepancy coefficients to generate the actual discrepancy coefficients $d^{(l)}_{\alpha'\beta'}$. The required update operations on the discrepancy coefficients, similar to that in (1) are given below:

$$d_{\alpha',\beta'} = \begin{cases} d^{(minl)}_{\alpha\beta} \times \tilde{d}^{(l)}_{\alpha',\beta'} + d^{(l)}_{\alpha\beta} \times \tilde{d}^{(minl)}_{\alpha',\beta'} & \text{if } l \neq minl \\ \bar{\delta}(\alpha') \times \tilde{d}^{(minl)}_{\alpha'-1,\beta'} & \\ +(x_i + x_i) \times \tilde{d}^{(minl)}_{\alpha',\beta'} & \text{if } l = minl \end{cases} \quad (2)$$

where $\bar{\delta}(\alpha')$, which takes values 0 and 1, indicates if a discrepancy coefficient corresponding to $X^{(\alpha'-1)}Y^{\beta'}$ exists. The crucial point is that the update of the tentative discrepancy coefficients in (2) leading to the next set of discrepancy coefficients can be done in parallel with **PU** in (1). Extending this idea further, we can compute a tentative set of discrepancy coefficients corresponding to all the constraints using the initialized candidate polynomials, at a low cost. These discrepancy coefficients appropriately updated, are used to update other discrepancy coefficients as in (2) and to update candidate polynomials as in (1). This leads to the reformu-

```
Initialization:
$Q_0(X,Y) = 1; Q_1(X,Y) = Y; \cdots ; Q_t(X,Y) = Y^t$
$Wdeg_0 = 0; Wdeg_1 = k-1; \ldots , Wdeg_t = t(k-1)$
Input: $(x_i, y_i, m_i),\ 0 \le i < s$
  Initial Discrepancy Coefficient Computation (IDCC):
  for $q = 1$ to $C$ do
    $d^{(l)}(0,q) = \mathrm{Coeff}[Q_l(X + x_{i_q}, Y + y_{i_q}), X^{\alpha_q} Y^{\beta_q}]$
  endfor
  Interpolation Iterations:
  for $p = 1$ to $C$ do (serially)
    Discrepancy Coefficient Scaling (DCS):
    DCS0: $minl = \mathrm{argmin}(Wdeg_l | d^{(l)}(p-1, C) \neq 0, l \le t)$
    DCS0: $Wdeg_{minl} \Leftarrow Wdeg_{minl} + 1$
    DCS1: $(0 \le l \le t$ and $l \neq minl)$
          $\lambda_p^{(l)} = d^{(l)}(p-1, C)/d^{(minl)}(p-1, C)$
    DCS2: $(l = minl)$
          $\lambda_p^{(l)} = x_{i_C}$
    Discrepancy Coefficient Update (DCU)
    for $q = p + 1$ to $C$ do (in parallel)
      DCU1: $(0 \le l \le t$ and $l \neq minl)$
            $d^{(l)}(p,q) = d^{(l)}(p, q-1) + \lambda_p^{(l)} d^{(minl)}(p, q-1)$
      DCU2: $(l = minl)$
            $d^{(l)}(p,q) = \bar{\delta}(\alpha_q) d^{(l)}(p-1, q)$
                $+ (\lambda_p^{(l)} + x_{i_q}) d^{(minl)}(p-1, q-1)$
    endfor
    Candidate Polynomial Coefficient Update (CPCU)
    for $q = C + 1$ to $C + \zeta(t + 1)$ do (in parallel)
      CPCU1: $(0 \le l \le t$ and $l \neq minl)$
             $d^{(l)}(p,q) = d^{(l)}(p-1, q) +$
                 $\lambda_p^{(l)} d^{(minl)}(p-1, q)$
      CPCU2: $(l = minl)$
             $d^{(l)}(p,q) = \bar{\delta}(\alpha_q) d^{(l)}(p-1, q-1)$
                 $+ \lambda_p^{(l)} d^{(minl)}(p-1, q)$
    endfor
  endfor
```

Fig. 3. Reformulated Interpolation Algorithm (RIA)

lated version of the interpolation algorithm. In order to reduce the number of multiplications, we rewrite (1) and (2) as:

$$\tilde{Q}_l(X,Y) = \begin{cases} Q_l(X,Y) + \\ \lambda_{\alpha\beta}^{(l)} \times Q_{minl}(X,Y) & \text{if } l \neq minl \\ XQ_l(X,Y) + \\ \lambda_{\alpha\beta}^{(l)} Q_{minl}(X,Y) & \text{if } l = minl \end{cases} \tag{3}$$

$$d_{\alpha',\beta'} = \begin{cases} \tilde{d}_{\alpha',\beta'}^{(l)} + \lambda_{\alpha\beta}^{(l)} \times \tilde{d}_{\alpha',\beta'}^{(minl)} & \text{if } l \neq minl \\ \bar{\delta}(\alpha') \times \tilde{d}_{\alpha'-1,\beta'}^{(l)} + \\ (\lambda_{\alpha\beta}^{(l)} + x_i) \times \tilde{d}_{\alpha',\beta'}^{(minl)} & \text{if } l = minl \end{cases} \tag{4}$$

where

$$\lambda_{\alpha\beta}^{(l)} = \begin{cases} \frac{d_{\alpha\beta}^{(l)}}{d_{\alpha\beta}^{(minl)}} & \text{if } l \neq minl \\ x_i & \text{if } l = minl \end{cases} \tag{5}$$

The inversion in (5) should not unduly alarm us as the inversion cost is amortized over a large number of multiplications. The pseudocode corresponding to the reformulated interpolation algorithm is shown in Fig. 3. The indexing used is explained in the next subsection.

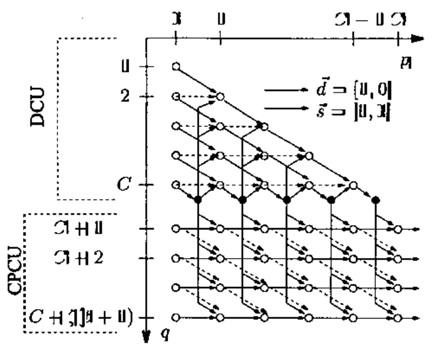

Fig. 4. Dependence graph for systolic interpolation

3.2. The Reformulated Interpolation Algorithm

The pseudocode in Fig. 3 uses an alternate indexing scheme equivalent to that of Fig. 2, in order to illustrate the systolic, regular nature of **RIA** . This indexing scheme is explained next.

The total number of constraints C associated with the ordered triples (x_i, y_i, m_i) with $0 \le i < s$ is $C = \sum_{i=0}^{s-1} \binom{m_i+1}{2}$. Using the technique of re-encoding presented in [5], it can be shown that the number of constraints reduces to $C = \sum_{i=k}^{s-1} \binom{m_i+1}{2}$, where for simplicity we have assumed that the $x_i's$ associated with the first k interpolation points are unique and are chosen for re-encoding.

In the **DCU** operations, we index the discrepancy coefficients as $d^{(l)}(p,q)$, where p is the iteration index and indicates the number of times the discrepancy coefficient has been updated. The index q which takes values from $p+1$ to C tracks the constraints that remain to be solved at the p^{th} iteration. This mapping of the index q is chosen so that at every iteration p, the discrepancy set $d^{(l)}(p-1, C)$ for $0 \le l \le t$ is used to update all other discrepancies and candidate polynomial coefficients. Hence, the value of q to which a particular constraint is mapped, is incremented with the iteration index p.

With each value of the index q at iteration p, we can associate a (x_i, y_i) and a particular value of α, β corresponding to $X^\alpha Y^\beta$. We can explicitly show this equivalence by rewriting the interpolation point as (x_{i_q}, y_{i_q}) and the constraint degree as α_q, β_q. In general, we drop the subscripts when there is no confusion.

A fundamental point of distinction between the **CSIA** and the **RIA** is with respect to discrepancy coefficient computation. In **CSIA**, the **DCC** computations use the updated current candidate polynomials to compute discrepancies. This discrepancy computation can be done only if all the coefficients of the candidate polynomials are known. Whereas in the **RIA** , the discrepancy values depend on the candidate polynomial coefficients only in the **IDCC** step, which requires only the initialized candidate polynomials. In the interpolation iterations, the discrepancy coefficients are completely independent of the candidate polynomial coefficients. Hence, we are at liberty to reduce the number of candidate polynomial coefficients iteratively computed, to as small a number as desired.

We use this property of the **RIA** to compute just as many candidate polynomial coefficients, as required in the minimal polynomial, by the factorization kernel. It can be shown that the number of minimal polynomial coefficients required in the factorization kernel using the techniques presented in [5], is $\zeta(t+1)$. ζ is given by $\zeta = 2\theta + \delta$, where θ is the number of symbol errors corrected

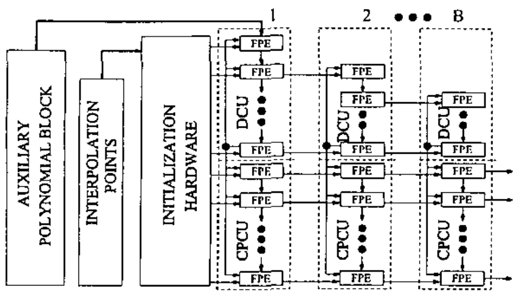

Fig. 5. Interpolation Processor Array

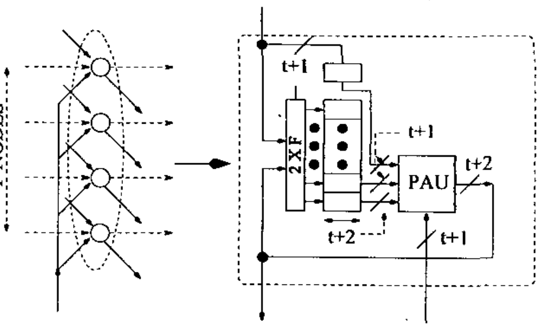

Fig. 6. Folded Processor Element (FPE)

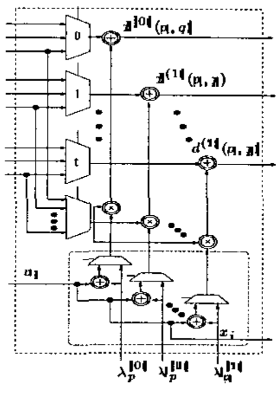

Fig. 7. Processor Arithmetic Unit (PAU)

per codeword using soft-decoding and δ is an integer value typically much less than 2θ. As we have $t+1$ candidate polynomials, we iterate a total of $\delta(t+1)^2$ candidate polynomial coefficients.

Another key observation used in the **RIA** is that, the candidate polynomial coefficients, can be viewed as discrepancy coefficients obtained by evaluating the candidate polynomials at $(0,0)$. This standpoint, which casts candidate polynomial coefficients as discrepancy coefficients, unveils the inherent regularity of operations across **DCU** and **CPCU** in the **RIA** . This perspective is maintained in the **CPCU** operations of Fig. 3, by denoting the candidate polynomial coefficients using $d^{(i)}(p,q)$. The index p is once again the iteration index and q, which now indexes the candidate polynomial coefficients runs from $C+1$ to $C+\delta(t+1)$ for every iteration. As the number of candidate polynomial coefficients iterated is invariant, the mapping of the coefficients to the index q in the **CPCU** step is also kept invariant to iterations.

4. SYSTOLIC ARCHITECTURES

In this section, we discuss the sequence of steps required to map the **RIA** to a hardware implementation. In subsection 4.1, we map the **RIA** to a dependence graph and use the dependence graph to generate a hardware implementation. Details of the processor cell forming the systolic architecture is discussed in subsection 4.2. Some details regarding the initialization block and the auxiliary polynomial block are given in subsection 4.3.

4.1. Mapping the RIA to an architecture

Graphs that show the dependence of the computations that occur in an algorithm are termed Dependence Graphs (**DG**)[8], [9]. The **DG** associated with the **RIA** is shown in Fig. 4. The computations that occur at a node (p,q) are as given in the **RIA** in Fig. 3. The dotted arrows show the dependency of a discrepancy coefficient on the discrepancy coefficient corresponding to its adjacent constraint. This can be seen in the **DCU2** and **CPCU2** operations. The **DCS** computations occur in the darkened nodes.

Generating systolic architectures from a **DG** requires specifying the architecture of each processor cell in the processor array, selecting an appropriate projection vector $(\vec{d})$, to map nodes to specific processor cells and selecting the scheduling vector $(\vec{s})$, to specify the time index of each computation.

In this paper, we generate two implementations of the **RIA** . The first implementation is a low-latency systolic interpolation architecture and the second implementation is a high-throughput systolic interpolation architecture. We envision that a low-latency architecture would find use in a scenario where the soft-decoder is triggered by a BMD decoder failure. The high-throughput architecture is useful in a soft-decoder operating as the default decoder for all received data vectors.

A generic block diagram for the interpolation processor array is shown in Fig. 5. The details of the *folded processor element* (**FPE**) is given in subsection 4.2. Some details regarding the auxiliary polynomial block and the initialization hardware are given in subsection 4.3. The low-latency architecture has only a single column of processor elements, whereas the number of columns in the high-throughput architecture is equal to the *block pipelining depth*, B.

For both the architectures, our choice of projection vector and schedule vector with reference to Fig. 4 is $\vec{d}=[1,0]$, $\vec{s}=[1,0]$. With this choice, the number of nodes per iteration drops linearly from $C+\delta(t+1)$ nodes to $\delta(t+1)$ nodes. The computations terminate at the end of C iterations. Another valid choice of vectors, $\vec{d}=[0,1]$, $\vec{s}=[1,1]$, requires $2C+\delta(t+1)$ iterations and the number of nodes increases from 1 upto C. In our implementations, we use $\vec{d}=[1,0]$, $\vec{s}=[1,0]$ as it minimizes the variation in the number of active nodes. This achieves higher hardware utilization.

For both the architectures, we need to map nodes to processor cells. In Fig. 4, we observe that the number of **DCU** nodes decrease as we progress along the q axis. Hence for the low-latency architecture, we desire a mapping technique in which the workload of each processing element decreases with the number of nodes. This requires processor elements which can vary the number of nodes mapped during run-time. An architecture for such a processor element is given in subsection 4.2.

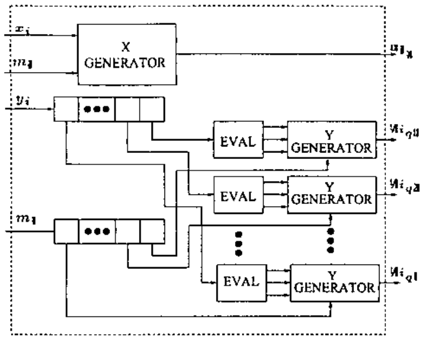

Fig. 8. Initialization Block

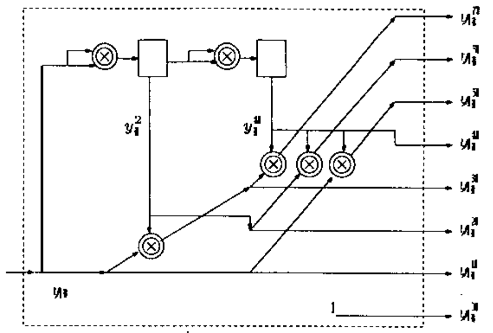

Fig. 9. Polynomial Evaluation Block

The high-throughput architecture with a block pipelining depth B, is obtained by partitioning the j axis of Fig. 4 into B parts. The reduction in the number of nodes is reflected in the smaller number of processing elements assigned to succeeding processor columns in Fig. 5. In this architecture, we map $\frac{C}{B}$ iterations to a processor cell column. We require each processor column to solve $\frac{C}{B}$ iterations with the same *block pipelining period*, which is the number of clock cycles required to complete $\frac{C}{B}$ iterations. As we have lesser variation of active nodes within $\frac{C}{B}$ iterations, simpler processing elements can be used.

4.2. Processor Cell Architecture

In this subsection, we discuss the architecture of the processing elements that form the systolic array. As the number of processor cells in a practical implementation is usually lesser than the number of active nodes in the **DG**, the folding technique [9], is used to map nodes to processor elements. A folded processor element (**FPE**) with F nodes of the **DG** mapped onto it is shown in Fig. 6. The storage registers have width $t+2$ to store the $t+1$ discrepancy coefficients and a corresponding x_i value. The $t+1$ wide register supplies the values required in the **DCU** and the **CPCU** operations when two adjacent nodes split across two **FPE** . The number of nodes mapped to a **FPE** is made run-time variable by using a $2 \times F$ switch, where F denotes the maximum number of nodes that can be mapped to the **FPE** . With iterations, as the number of nodes mapped to a **FPE** decrease, the number of clock cycles required to process a constraint also decreases. The Processor Arithmetic Unit (**PAU**), performs the **DCU** and the **CPCU** arithmetic operations. The details of the **PAU** is shown in Fig. 7. The adder-multiplexer sub-block modifies the $\lambda_q^{(l)}$ coefficients as in **DCU2.**

The critical path of the interpolation processor array for the low-latency architecture and for the high-throughput architecture occurs in the **FPE** . It consists of the path through the **PAU** which comprises of a $t+1$:1 multiplexer, followed by an adder and a multiplier. This is followed by the $2 \times F$ switch, formed using a 2:1 multiplexer and a 1:F de-multiplexer.

The critical path of the high-throughput architecture can be further reduced with the reasonable assumption that the folding factor remains the same over the $\frac{C}{B}$ iterations in a processor column. We then replace the $2 \times F$ switch in the **FPE** with a 2:1 multiplexer.

4.3. Initialization hardware

In this subsection, we give some details of the initialization hardware and the auxiliary polynomial block. As stated earlier, the re-encoding transformation presented in [5], reduces the number of interpolation constraints to be solved. These transformations also modify the candidate polynomial initialization given in **CSIA** . The modified initialization is $Q_0(X,Y) = h_0(X)$, $Q_1(X,Y) = h_1(X)Y$; $\cdots$; $Q_t(X,Y) = h_t(X)Y^t$, where $h_0(X), h_1(X), \cdots, h_t(X)$, are termed the *auxiliary polynomials* [7]. Hence in the **IDCC** step, we evaluate $h_l(X + x_i)$ for $0 \leq l \leq t$ and for all x_i with non-trivial interpolation points. The simplifications in evaluating $h_l(X + x_i)$ will be presented as part of a forthcoming paper. For a fair comparison in this paper, we neglect the hardware resources required to compute $h_l(X + x_i)$ both in the proposed architectures and in the previous architecture.

A block diagram corresponding to the initialization block is shown in Fig. 8. The **X Generator** is used to generate the initial mapping of the x_{i_q} for $0 \leq q \leq C$. The **X Generator** essentially repeats the x_i symbol $\binom{m_i+1}{2}$ times for each interpolation point (x_i, y_i). The details of the **EVAL** block which computes $(Y + y_i)^t$ is shown in Fig. 9. These powers appropriately repeated using the **Y Generator** are sent to the processing cells to complete the **IDCC** step, thus initializing the processor array. We would like to point out that a generalized form of Fig. 9 can be used to evaluate any univariate polynomial shifted by a field element.

5. COMPARISON WITH PREVIOUS ARCHITECTURES

In this section, we compare the hardware implementations of the systolic architecture with non-systolic architectures such as the one in [7]. In order to compare architectures, we fix the total number of multiply-accumulate (MAC) units used in an implementation, denoted by N_{MAC}. The complexity of interpolation is determined by the total constraints to be solved C, or equivalently by the *average multiplicity* m_{avg}, as defined in [7]. For the architecture of [7], it is feasible to estimate decoding latency with the assumption that the number of available MAC units is a multiple of $(t+1)^2$. This multiple is termed the *parallelization factor*. The parallelization factor for **DCC** and **PU** is denoted P_D and P_U respectively. Hence, we obtain the relation: $P_D(t+1)^2 + P_U(t+1)^2 = N_{MAC}$. We denote by X_{avg}, the average over the whole interpolation process, of the maximum X degree of the candidate polynomials at each iteration. With the reasonable assumption that the maximum X degree in the candidate polynomials increases linearly to the final value, we fix X_{avg} at half the maximum value. The number of clock cycles required for polynomial update is : $D_{PU} = C \times \left\lceil \frac{X_{avg}}{P_U - 1} \right\rceil$. The parallelization is only by a factor $P_U - 1$ as $(t+1)^2$ MACS are utilized for discrepancy coefficient update.

Table 1. COMPARISON OF SYSTOLIC V/S NON-SYSTOLIC ARCHITECTURES

Implementation	Storage (Bytes)	Latency/Block pipelining period (cycles)	[255, 239] RS example Latency (cycles)	[255, 239] RS example Throughput gain
[7]	$\approx C(t+2)$	$\frac{A}{E}\left[1+\sqrt{\frac{D}{A}}\right]^2+G$	≈ 3000	-
Low-latency	$\zeta(t+1)^2+C(t+2)$	$\sum_{q=0}^{C} F^{(i)}$	≈ 1975 (-34 %)	52 %
High-throughput	$B\zeta(t+1)^2+\frac{C}{2}(t+2)(B+1)$	$\frac{C}{N_{MAC}}\left[\zeta(t+1)^2+\frac{(t+1)C}{2}\left(1+\frac{1}{B}\right)\right]$	1785×5 (+200 %)	68 %

It can be shown that the number of clock cycles required for the **DCC** operation is:

$$D_{DPC} = \sum_{i=k}^{s-1}\left(\frac{X_{avg}}{P_D} + \frac{(t+1)m_i}{P_D} + m_i\right) \qquad (6)$$

The summation in (6) is over the interpolation points that remain to be solved after re-encoding. Since the interpolation points which are not re-encoded are usually unreliable, the average multiplicity $\hat{m}_{avg}$, among these points is about half the average multiplicity m_{avg}. With this assumption, (6) can be reduced to:

$$D_{DPC} = \frac{2C}{\hat{m}_{avg}^2}\frac{X_{avg}}{P_D} + \frac{2C(t+1)}{\hat{m}_{avg}P_D} + \frac{2C}{\hat{m}_{avg}} \qquad (7)$$

The minimum total interpolation latency obtained by optimizing over the parallelization factors, is given in Table 1. A, D, E, G are constants given by $A = \frac{2C}{\hat{m}_{avg}}(\frac{X_{avg}}{\hat{m}_{avg}} + t + 1)$, $D = X_{avg} \times C$, $E = \frac{N_{MAC}}{(t+1)^2} - 1$, $G = \frac{2C}{\hat{m}_{avg}}$ and the optimum $P_D = \frac{E}{1+\sqrt{\frac{D}{A}}}$. The critical path in this architecture occurs in the **PU** unit. It comprises of a $P_U - 1 : 1$ multiplexer and the $1 : P_U - 1$ de-multiplexer used to route data into the parallel computation units and the critical path along the parallel computation unit, which is identical to that of the **PAU** , and comprises a $2 : 1$ multiplexer, a multiplier, and an adder. The critical path of this architecture is larger than that of the systolic architecture as the $1 : P_U - 1$ de-multiplexer is usually larger than the corresponding 2:1 multiplexer used in **FPE** .

We now consider the decoding latency of the low-latency systolic architecture. Let $F^{(q)}$ be the folding factor at the q^{th} iteration. Since the total number of MAC units is fixed at N_{MAC}, we get $F^{(q)} = \left\lceil\frac{1}{N_{MAC}}(C-q)(t+1) + \zeta(t+1)^2\right\rceil$. An expression for the total interpolation latency is given in Table 1.

For the high-throughput, block pipelined systolic architecture, the block pipelining period as a function of the block pipelining depth B is given in Table 1.

We now evaluate the formulae given in Table 1 for a [255, 239] RS soft-decoder operating with $m_{avg} = 3$. Since we use the re-encoding technique [5], the constraints to be satisfied by interpolation reduce from $n\binom{m_{avg}+1}{2}$ to around $(n-k)\binom{m_{avg}+1}{2}$ at a high channel SNR. For our example, we assume moderate SNRs for which $C \approx 500$ after re-encoding. The maximum number of symbol errors that we attempt to correct is $\theta = 13$ and hence we use $\zeta = 30$. The high throughput architecture results are obtained at $B = 5$. The storage requirements of each of the implementations is also tabulated.

6. CONCLUSIONS

In this paper, we have presented systolic architectures obtained via the reformulated interpolation algorithm. Details of a low-latency and a high throughput systolic architecture have been presented and compared with previously reported implementations. The systolic framework presented in this paper can be easily extended to encompass the factorization kernel. Jointly allocating fixed hardware resources to a systolic interpolation and factorization kernel in accordance with relevant optimization criteria would be a subject for future work.

7. REFERENCES

[1] M. Sudan, "Decoding of Reed-Solomon codes beyond the error-correction bound," *Journal of Complexity*, vol. 13, no. 1, pp. 180-193, March 1997.

[2] V. Guruswami and M. Sudan, "Improved decoding of Reed-Solomon and algebraic-geometric codes," *IEEE Trans. Info. Theory* , vol. 45, pp. 1757-1767, Sept. 1999.

[3] R. Koetter and A. Vardy, "Algebraic soft-decision decoding of Reed-Solomon codes," Submitted to *IEEE Trans. Info. Theory*, May 31, 2000.

[4] R. E. Blahut, *Algebraic Codes for Data Transmission*, Preprint., Feb 2001.

[5] R. Koetter, J. Ma, A. Vardy and A. Ahmed, "Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes," *Accepted at the IEEE Symp. on Info. Theory*, 2003.

[6] W. J. Gross, F. R. Kschsichang, R. Koetter and P. G. Gulak, "A VLSI architecture for interpolation in soft-decision list decoding of Reed-Solomon codes," *Proc. of IEEE Workshop on Signal Processing Systems*, Oct. 2002.

[7] A. Ahmed, R. Koetter, and N.R. Shanbhag, " VLSI architectures for soft-decision decoding of Reed-Solomon codes," *Submitted to the IEEE Trans. VLSI Systems*, Feb 2003. (http://tesla.csl.uiuc.edu/ koetter/publications.html)

[8] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, Englewood Cliffs NJ, 1987.

[9] K. K. Parhi, *VLSI Digital Signal Processing Systems*, John Wiley and Sons, 1999.

[10] D. V. Sarwate and N. R. Shanbhag, "High-speed architectures for Reed-Solomon decoders," *IEEE Trans. VLSI Systems*, vol. 9, pp. 641-655, Oct. 2001.

[11] E. R. Berlekamp, G. Seroussi, and P. Tong, "A hypersystolic Reed-Solomon decoder," in *Reed-Solomon Codes and Their Applications*, S. B. Wicker and V. K. Bhargava (eds.), IEEE Press, Piscataway NJ 1994.

[12] G. L. Feng and K. K. Tzeng, "A generalization of the Berlekamp-Massey algorithm for multi-sequence shift-register synthesis with applications to decoding cyclic codes," *IEEE Trans. Info. Theory*, vol. 37, pp. 1274-1287, Sept. 1991.