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 [233, 229] 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.

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, d] 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 ≤ i ≤ s. The ordered pair (x_i, y_i) is termed an interpolation point and is a set of finite field elements in some GF(q_i). 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, Y + x_i), the coefficients of all terms X^dY^e with d + e < m_i are 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 ≤ i ≤ s and corresponding multiplicities m_i, compute the minimal polynomial Q(X, Y), which is the non-trivial polynomial of minimum 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 f(X) = f(X) with degree f(X) < k. Each of the univariate polynomials f(X) corresponds to a valid data polynomial.
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 \((a_i, y_i)\) with multiplicity \(a\), imposes \(\ell\) constraints, where each constraint corresponds to a unique \((\alpha,\beta)\) which satisfies \(\alpha + \beta < m\). For each of these constraints, the coefficient of \(X^\alpha Y^\beta\) in \(Q_{i}(X, Y) + a_i + y_i\) for each of the candidate polynomials is determined in the DCC step. The coefficients \(d_{\alpha,\beta}^{(i)}\), 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 \([1, \ell-1]\) weighted degree with a non-zero discrepancy and denotes its index \(\text{mind}\). The step PU1 is analogous to the Berlekamp-Massey update equations, with \(Q_{\text{mind}}(X, Y)\) playing the role of the scratch polynomial. In Step PU2, the polynomial with index \(\text{mind}\) is multiplied with \(X + a_i\). This forces the polynomial to satisfy the current constraint but also results in an 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:**

\[
\begin{align*}
Q_{1}(X, Y) = y_1, & \quad Q_{i+1}(X, Y) = Y^{t_i-1} \cdot Q_i(X, Y) \\
W_1(y_1,m_1) = 0, & \quad W_{i}(y_i,m_i) = W_{i-1}(y_i,m_i) - \delta_{m_i}^{(i)}
\end{align*}
\]

**Input:** \((x_i, y_i, m_i); 1 \leq i < s\)

**Discrepancy Coefficient Computation (DCC):**

\[
\delta_{\alpha,\beta}^{(i)} = \text{Coeff}[Q_i]X^\alpha + y_i, \quad Y^{t_i}\]

**Polynomial Update (PU):**

- **PU0:** \(\text{mind} = \arg\min(W_{\text{deg}}Q_k, 4 \leq k \leq 3)\)
- **PU1:**
  \[
  \begin{align*}
  Q_{1}(X, Y) & = d_{0,0}^{(i)} \times Q_{\text{mind}}(X, Y) \\
  d_{0,0}^{(i)} & = \text{mind}
  \end{align*}
  \]
- **PU2:**
  \[
  \begin{align*}
  Q_{\text{mind}}(X, Y) & = Q_{\text{mind}}(X, Y) + [X + x_i] \\
  W_{\text{deg}}Q_{\text{mind}} & = W_{\text{deg}}Q_{\text{mind}} + 1
  \end{align*}
  \]

**Output:** \(Q_{\text{mind}}(X, Y), \text{mind} = \arg\min(W_{\text{deg}}Q_k, 4 \leq k \leq 3)\)

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

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

3.1. Principal idea driving reformulation

Let \(d_{\alpha,\beta}^{(i)}\) with \(0 \leq 1 \leq \ell\) be the set of discrepancy coefficients computed at a constraint corresponding to a particular interpolation point \((x_i, y_i)\) with \(1 \leq \ell \leq m_i\). As in the pseudo-code of Fig. 2, the PU equations are given by:

\[
\begin{align*}
Q_{\text{mind}}(X, Y) & = d_{0,0}^{(i)} \times Q_{\text{mind}}(X, Y) \\
Q_{\text{mind}}(X, Y) & = Q_{\text{mind}}(X, Y) + [X + x_i]
\end{align*}
\]

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

\[
\begin{align*}
\delta_{\alpha,\beta}^{(i)} & = \delta_{\alpha,\beta}^{(i)} + d_{\alpha,\beta}^{(i)} \times d_{\alpha,\beta}^{(i)} \\
\delta_{\alpha,\beta}^{(i)} & = \delta_{\alpha,\beta}^{(i)} + [X + x_i] \times d_{\alpha,\beta}^{(i)}
\end{align*}
\]

where \(\delta_{\alpha,\beta}^{(i)}\), which takes values 0 and 1, indicates if a discrepancy coefficient corresponding to \(X^{t_i-1}Y^{t_i}\) 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:
\[ Q_0(X, Y) = 1, Q_1(X, Y) = Y, \ldots, Q_d(X, Y) = Y^d \]
\[ W \equiv \delta = 3, W \delta \equiv \delta = k - l, \ldots, W \delta^n \equiv \delta = k - l^n \]
Input: \([a_1, a_2, m_1], 1 \leq i \leq s \]

Initial Discrepancy Coefficient Computation (IDCC):
for \( s = 1 \) to \( 3 \) do
\[ d^{(1)}(i, q) = \text{Coeff}[X^q Y^{a_q}] + y_q X^q + X^{a_q} Y^{q} \]
endfor

Interpolation Iterations:
for \( p = 1 \) to \( t \) do (serially)
Discrepancy Coefficient Scaling (DCS):
DCS0: \( \text{mind} = \text{argmin} W \delta^{(0)}(p, \delta \neq \text{mind}) \)
DCS1: \( (0 \leq i \leq l \text{ and } \delta \neq \text{mind}) \)
\[ \lambda_{(i)}^{(1)} = d^{(1)}(i - 2, 3) \]
DCS2: \( (i = \text{mind}) \)
Discrepancy Coefficient Update (DCU)
for \( q = 1 \) to \( t \) do (in parallel)
DCU1: \( (1 \leq i \leq l \text{ and } \delta \neq \text{mind}) \)
\[ d^{(1)}(i, q) = d^{(1)}(i, q - 1) + \lambda_{(i)}^{(1)} X^{(a_q + i)} Y^{(q - 1)} \]
DCU2: \( (i = \text{mind}) \)
\[ d^{(1)}(i, q) = \tilde{d}(i, q) + \lambda_{(i)}^{(1)} X^{(a_q + i)} Y^{(q - 1)} \]
endfor
Candidate Polynomial Coefficient Update (CPCU)
for \( q = 1 \) to \( t \) do (in parallel)
CPCU1: \( (1 \leq i \leq l \text{ and } \delta \neq \text{mind}) \)
\[ d^{(1)}(i, q) = d^{(1)}(i, q - 1) + \lambda_{(i)}^{(1)} X^{(a_q + i)} Y^{(q - 1)} \]
CPCU2: \( (i = \text{mind}) \)
\[ d^{(1)}(i, q) = \tilde{d}(i, q) + \lambda_{(i)}^{(1)} X^{(a_q + i)} Y^{(q - 1)} \]
endfor
Fig. 3. Reformulated Interpolation Algorithm (RIA)

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 \([a_1, a_2, m_1]\) with \( 0 \leq i \leq s \) is
\[ C = \sum_{l=1}^{C} \sum_{p=1}^{(l-1)} \sum_{q=1}^{(l-1)} \sum_{s=1}^{(l-1)} \]
Using the technique of re-encoding presented in [5], it can be shown that the number of constraints reduces to
\[ \sum_{l=1}^{C} \sum_{p=1}^{(l-1)} \sum_{q=1}^{(l-1)} \sum_{s=1}^{(l-1)} \]
where for simplicity we have assumed that the \( s \) is associated with the first \( i \) interpolation points are unique and are chosen for re-encoding.

In the DCU operations, we index the discrepancy coefficients as \( d^{(1)}(i, q) \), 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 \( 0 \) to \( t \) 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 \( d^{(1)}(p - 1, C) \) for \( 0 \leq i \leq t \) 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 \([a_1(q), a_2(q)]\) and a particular value of \( \alpha, \beta \) corresponding to \( XPY^q \). We can explicitly show this equivalence by rewriting the interpolation point as \((x_{i1}, y_{i2})\) and the constraint degree as \( a_1(q) \beta \).

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 initial candidate polynomial coefficients 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 \( \sum_{l=1}^{C} \sum_{p=1}^{(l-1)} \sum_{q=1}^{(l-1)} \sum_{s=1}^{(l-1)} \)
where \( \gamma = 2 \theta + \delta \), where \( \theta \) is the number of symbol errors corrected.
4. SYSTEM 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 are 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 \( j_p, j_q \) are as given in the RIA in Fig. 3. The dotted arrows show the dependency of a discrepancy coefficient corresponding to its adjacent constraint. This can be seen in the DCU and CPCU operations of Fig. 3, by denoting the candidate polynomial coefficients using \( \delta^H(j_p, j_q) \). The index \( p \) is once again the iteration index and \( q \), which now indexes the candidate polynomial coefficients runs from \( C + 1 \) to \( C + \| J \| \) 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.

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 \( [1, \| J \| ] \). 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 \( \delta^H(j_p, q) \). The index \( p \) is once again the iteration index and \( q \), which now indexes the candidate polynomial coefficients runs from \( C + 1 \) to \( C + \| J \| \) 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.

Per codeword using soft-decoding and \( \delta \) is an integer value typically much less than 29. As we have \( t + 1 \) candidate polynomials, we iterate a total of \( 3(t + 1)^p \) candidate polynomial coefficients.

Fig. 5. Interpolation Processor Array

Fig. 6. Folded Processor Element (FPE)

Fig. 7. Processor Arithmetic Unit (PAU)

4.2. Generating systolic architectures from a dependence graph

Generating systolic architectures from a DG requires specifying the architecture of each processor cell in the processor array, selecting an appropriate projection vector \( \bar{J} \), to map nodes to specific processor cells and selecting the scheduling vector \( \bar{S} \), 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, \( D \).

For both the architectures, our choice of projection vector and the number of nodes per iteration drops linearly from \( C + \| J \| \) to \( [1, \| J \| ] \) nodes. The computations terminate at the end of \( C \) iterations. Another valid choice of vectors, \( \bar{J} = [1, \| J \| ], \bar{S} = [1, \| J \| ] \), requires \( 2C + \| J \| \) iterations and the number of nodes increases from 1 up to \( \| J \| \). In our implementations, we use \( \delta = [1, 0], \bar{S} = [0, 0] \) 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.
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 less 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 \( t + 3 \) to store the \( 3:1 \) discrepancy coefficients and a corresponding \( x_3 \) value. The \( t + 8 \) 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 \( 2 \times 1 \) 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 \( \lambda_{PCU}^{MU} \) 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 \( t + 1 \) 1:1 multiplier, followed by an adder and a multiplier. This is followed by the \( 2 \times F \) switch, formed using a 2:1 multiplexer and a 1:1 demultiplexer.

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 \( \frac{B}{2} \) iterations in a processor column. We then replace the \( 3:1 \) 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 \( Q_{Ij}(X,Y) = 0 \ldots \). The details of the EVAL block which computes \( \lambda_{PCU}^{MU} \) is shown in Fig. 9. The 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 \( m_{mac} \), 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 \( t + 1 \). This multiple is termed the parallelization factor. The parallelization factor for DCC and PU is denoted \( P_D \) and \( P_P \) respectively. Hence, we obtain the relation: \( P_D (t + 1)^{P_D} + P_P (t + 1)^{P_P} = N_{MAC} \). We denote by \( X_{mac} \), 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 \( X_{mac} \) at half the maximum value. The number of clock cycles required for polynomial update is: \( D_{DCC} = \frac{C}{N_{MAC}} \). The re-encoding is only by a factor \( P_D - 1 \) as \( t + 1 \) MACs are utilized for discrepancy coefficient update.
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


### Table 1. COMPARISON OF SYSTOLIC VS NON-SYSTOLIC ARCHITECTURES

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Storage (Bytes)</th>
<th>Latency/Block pipelining period (cycles)</th>
<th>255, 330 RS example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td></td>
<td>( \approx C (t + 3l) )</td>
<td></td>
</tr>
<tr>
<td>Low-latency</td>
<td></td>
<td>( \approx \frac{1}{2} (1 + \sqrt{8}) + C )</td>
<td>( \approx 3000 )</td>
</tr>
<tr>
<td>High-throughput</td>
<td></td>
<td>( \frac{1}{2} (1 + 1) + (C + 1) )</td>
<td>( \approx 1975 )</td>
</tr>
</tbody>
</table>

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

\[
D_{DCC} = \sum_{i=1}^{m} \left( \frac{X_{m+1}}{P_{l}} + \frac{(t+1) + C}{P_{0}} + \frac{m_{[m]}}{P_{0}} \right)
\]

(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 \( m_{[m]} \), among these points is about half the average multiplicity \( m_{avg} \). With this assumption, (6) can be reduced to:

\[
D_{APC} = \frac{3}{2} \frac{X_{m+1}}{P_{l}} + \frac{3}{2} \frac{C (t + 1)}{P_{0}} + \frac{X_{m+1}}{P_{0}} + \frac{m_{[m]}}{P_{0}}
\]

(7)

The minimum total interpolation latency obtained by optimizing over the parallelization factors, is given in Table 1. \( D_{l}, D_{l}, \eta_{l}, \eta_{l} \) are constants given by \( D_{l} = \frac{3}{2} \frac{X_{m+1}}{P_{l}} + \frac{t + 1}{P_{0}}, \eta_{l} = \frac{3}{2} \frac{X_{m+1}}{P_{l}} \) and \( \eta_{l} = \frac{3}{2} \frac{X_{m+1}}{P_{l}} \) and the optimum \( P_{0} \) is given by: \( \eta_{l} \). The critical path in this architecture occurs in the PU unit. It comprises of a \( P_{0} - t - 1 \) multiplexer and the \( t + 1 \) \( P_{1} - \eta_{l} \) 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 \( t \) : 1 multiplexer, a multiplier, and an adder. The critical path of this architecture is larger than that of the corresponding \( 2:1 \) multiplexer used in the FPE.

We now consider the decoding latency of the low-latency systolic architecture. Let \( E = \frac{D_{l}}{D_{l}} \) be the folding factor at the \( P_{l} \) iteration. Since the total number of MAC units is fixed at \( N_{MAC} \), we get \( F = \frac{E}{E} \frac{(t + 1)}{P_{0}} \frac{(t + 1)}{P_{0}} \frac{(t + 1)}{P_{0}} \). 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 formulas given in Table 1 for a [255, 330] 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 \( m_{[m] + 1} \) to \( m_{[m]} \) and \( 8(C + 1) \) at a high channel SNR. For our example, we assume moderate SNRs for which \( C = 12 \) after re-encoding. The maximum number of symbol errors that we attempt to correct is \( t = 13 \) 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.