Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 313479, 14 pages
doi:10.1155/2010/313479

Research Article

Reconfigurable Hardware Implementation of a Multivariate Polynomial Interpolation Algorithm

Rafael A. Arce-Nazario,[1] Edusmildo Orozco,[1] and Dorothy Bollman[2]

[1] *Department of Computer Science, University of Puerto Rico, Río Piedras, PR 00924, Puerto Rico*
[2] *Department of Mathematical Sciences, University of Puerto Rico, Mayagüez, PR 00681, Puerto Rico*

Correspondence should be addressed to Rafael A. Arce-Nazario, rafael.arce@upr.edu

Received 2 March 2010; Revised 26 July 2010; Accepted 26 October 2010

Academic Editor: Viktor K. Prasanna

Multivariate polynomial interpolation is a key computation in many areas of science and engineering and, in our case, is crucial for the solution of the reverse engineering of genetic networks modeled by finite fields. Faster implementations of such algorithms are needed to cope with the increasing quantity and complexity of genetic data. We present a new algorithm based on Lagrange interpolation for multivariate polynomials that not only identifies redundant variables in the data and generates polynomials containing only nonredundant variables, but also computes exclusively on a reduced data set. Implementation of this algorithm to FPGA led us to identify a systolic array-based architecture useful for performing three interpolation subtasks: Boolean cover, distinctness, and polynomial addition. We present a generalization of these tasks that simplifies their mapping to the systolic array, and control and storage considerations to guarantee correct results for input sequences longer than the array. The subtasks were modeled and implemented to FPGA using the proposed architecture, then used as building blocks to implement the rest of the algorithm. Speedups up to 172× and 67× were obtained for the subtasks and complete application, respectively, when compared to a software implementation, while achieving moderate resource utilization.

1. Introduction

Recent years have seen a significant increase in methods and tools to collect genetic data from which important information can be extracted using a number of techniques [1]. For instance, microarray data collected at various steps in organism development can help geneticists understand its developmental process and response to environmental stimuli [2]. Several models such as ordinary differential equation models [3], continuous models [4], stochastic models [5], and discrete models, of which Boolean models [6] are special cases, have been proposed. Essentially, each node represents the expression level of a gene (or genes) of interest at a time point. A directed edge from node a to node b symbolizes the influence of gene(s) in a to the gene(s) in b.

Our research group focuses on multivariate finite field gene network (MFFGN) models, in which multiple genes are monitored at each time step and their expression levels are discretized to a predefined set of values $\{0, 1, 2, \ldots, p-1\}$, where p is a prime number, that is, the expression level of each gene is an element of the prime field $\mathbb{Z}_p$ [7, 8]. One way to deduce gene interaction from these models is to solve the reverse engineering problem, that is, find functions that describe the network's state transitions. An important step in the solution of this problem is determining an interpolating polynomial for the points by using such methods as a multivariate version of Lagrange's interpolation formula [8]. Polynomial interpolation over finite fields also has applications in error correcting codes and is a major building block of many numerical methods [9, 10].

Fast algorithms and implementations are needed to sustain rapidly increasing bioinformatics computational demands [11]. Reconfigurable logic represents an attractive platform for the high performance implementation of finite field algorithms, with many designs achieving significant speedups over their software equivalents. However, despite the availability of tools to ease the hardware design flow, there is still a notable gap between the bioinformatics and

hardware experts [12]. Our group developed a reconfigurable logic implementation of a multivariate polynomial interpolation algorithm suitable for reverse engineering in genetic networks. Rather than striving for a specialized punctual design, we used this opportunity to identify and develop common structures that might be of use to other researchers and applications. Well-documented and parameterizable components that perform functionalities that are common in many algorithms will alleviate the effort spent on new implementations.

In this paper, we discuss the interpolation algorithm and our proposed architecture. We emphasize several computational substructures that appear repeatedly throughout the design and which can be implemented effectively using a hardware systolic array-based architecture. These tasks are of the type where we perform a certain reduction or rearrangement of a sequence of elements from multivariate polynomials or boolean expressions. The systolic array concurrently manages data receipt and parallel processing, making it an amenable structure when dealing with streamed data. The simplicity of the array cells, storage, and control unit, allows the instantiation of multiple cells while maintaining competitive clock frequencies, thus achieving high performance. Several tasks critical to interpolation were modeled and implemented to FPGA using the proposed architecture, obtaining speedups up to 172× when compared to a software implementation, while achieving low resource utilization. These implementations were used as components to develop a complete, high-performance multivariate polynomial interpolation methodology on hardware.

Paper Outline. Section 2 discusses the interpolation methodology, while Section 3 details data representation for the implementation. Section 4 describes the hardware implementation in general. Sections 5 and 6 describe an algorithmic generalization for the reduction tasks and present the proposed hardware architecture. Section 7 explains the rest of the implementation blocks. Section 8 reports the results of FPGA implementations, both of reduction tasks and the complete implementation. Section 9 provides our conclusions.

2. Interpolation Methodology

Discrete models, in particular finite field models, have been proposed for regulatory processes such as genetic networks [7] and biochemical networks [13]. In the setting of a finite field model a genetic network can be seen as a finite system whose states are governed by the iterations of a tuple of polynomials $(P_1, P_2, \ldots, P_n)$, where n is the number of genes.

Given a sequence of k n-tuples of values from a set F, representing the states of n genes at times t_0 through t_{k-1}, the reverse engineering problem seeks to find a function $f : F^n \to F^n$ for which $f(s_0) = s_1$, $f(s_1) = s_2, \ldots, f(s_{k-2}) = s_{k-1}$.

When F is a finite field, function f can be represented by a tuple of polynomials $(P_1, P_2, \ldots, P_n)$ where $P_i : F^n \to F$ for each $i = 1, 2, \ldots, n$. As pointed out in [8] such polynomials can be found by the following variation of Lagrange's interpolation formula:

$$P_i(x) = \sum_{m=0}^{k-1} f_i(a_m) \prod_{j=0, j\neq m}^{k-1} \frac{x_{l_{jm}} - a_{jl_{jm}}}{a_{ml_{jm}} - a_{jl_{jm}}}, \quad (1)$$

where each a_m denotes an n-tuple and a_{ml} denotes the lth component of a_m. Given a_j and a_m, where $a_j \neq a_m$, l_{jm} is the first component in which a_j and a_m differ.

In the context of genetic networks modeled by polynomials over a finite field, the resulting polynomials give a sense to the biologist about the interactions between genes. Nevertheless, results from (1) may contain *redundant* variables even after algebraic simplifications.

Table 1: Table for $f : \mathbb{Z}_3^3 \to \mathbb{Z}_3$.

x_1	x_2	x_3	f
0	1	2	0
1	1	2	0
1	2	0	1
0	2	1	2
1	1	0	2

Example 1. Let us consider the function $f : \mathbb{Z}_3^3 \to \mathbb{Z}_3$ which is incompletely specified by Table 1.

Using (1), an interpolating polynomial for f is given by $P = x_1^2x_2x_3 + x_1^2x_2^2 + x_1^2x_3 + 2x_1^2x_2 + x_1x_2 + 2x_2 + 2x_1^2 + 2x_1 + 1$. This polynomial depends on x_1,x_2, and x_3 and cannot be further simplified by any algebraic means. However, the polynomial $P' = 2x_2 + 2x_2x_3$ also interpolates f at the given points but depends only on variables x_2 and x_3. In this context, x_1 is a redundant variable, since we found another interpolating polynomial for f that does not include x_1 in its expression.

Redundant variables are undesirable as they introduce complexity into the polynomials without adding information valuable to the biologist. Furthermore, empirical data suggest that genetic networks are sparsely connected [14]. Redundant variables can be identified using Sasao's algorithm for the detection of dependent variables in incompletely specified multiple valued functions [15].

2.1. Essential Variables and Bases. For any set S and any positive integer n, we denote each $s \in S^n$ by $(s_1, s_2, \ldots, s_n)$. Let $U = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables. For any set of variables $X = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\} \subset U$ and any $s \in S^n$, we define $\text{proj}_X s = \{s_{i_1}, \ldots, s_{i_m}\}$. For instance, if $U = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $X = \{x_2, x_3, x_6\}$, and $s = (1, 0, 3, 2, 0, 1)$, the projection of s on X, $\text{proj}_X s$, is $\{0, 3, 1\}$.

Let $A = \{0, 1, \ldots, a-1\}$ and $B = \{0, 1, \ldots, b-1\}$ where a and b are integers, $a, b \geq 2$ and let $f : D \to B$, where $D \subset A^n$.

Table 2

x_1	x_2	x_3	x_4	x_5	f
0	2	1	2	3	1
0	2	1	3	1	1
2	1	1	0	1	1
0	1	1	2	0	2
2	1	2	1	0	2
2	1	2	0	1	2
2	1	2	2	1	2
0	1	2	1	1	3
2	2	2	2	2	3

We say that a variable x_i is *essential* in f or *depends on* f if there exist $c, d \in D$ such that $c_i \neq d_i$, $c_j = d_j$ for all $j \neq i$, and $f(c) \neq f(d)$. A set of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$ is a *basis* for f if (1) for every $c, d \in D$, $\text{proj}_X c = \text{proj}_X d$ if and only if $f(c) = f(d)$ and (2) there is no other set of variables that properly contains X with this property.

Lemma 1. *If x is an essential variable and X is a basis, then $x \in X$.*

Proof. Let x be an essential variable. Then there exists $c, d \in D$ and i such that $c_i \neq d_i$, $\text{proj}_{U-\{x_i\}} c = \text{proj}_{U-\{x_i\}} d$ and $f(c) \neq f(d)$. Let X be any basis. Then $X \not\subset U - \{x_i\}$, but $x \subset U$. Hence $x_i \in X$. □

For any function $f : D \to B$ and any basis X, f can be expressed solely in terms of the variables of X. For any basis X, any variable $x \notin X$ is *redundant* (with respect to X.)

Our goal is to determine interpolating polynomials over finite fields in terms of the variables of any of its bases. To this end, let us first review an algorithm of Sasao [15] which determines all bases for any multiple valued function, not just those over finite fields.

Let $f : D \to B$, where $D \subset A^n$. For each $i = 0, 1, \ldots, b-1$, let $S_{f,i} = \{s \in D \mid f(s) = i\}$.

The set of variables appearing in each disjunct of R constitutes a basis.

Thus, the algorithm consists of two stages. First, determine R, which requires $O(b^2)$ steps and second, convert R to disjunctive normal form, which requires $O(2^n)$ steps.

The following lemma, whose proof is immediate, is useful in simplifying expressions such as $x_1 \wedge x_1 x_2 = x_1$ and $x_1 \vee x_1 x_2 = x_1$ in the computation of R.

Lemma 2. *Let $S = \{x_{i_1}, x_{i_2}, \ldots, x_{i_d}\}$ be a subset of the variables $\{x_1, x_2, \ldots, x_n\}$, let E be the disjunction of the variables in S and let F be a disjunction of some subset of S. Also let G be the conjunction of some subset of S. Then*

(a) $E \wedge F = F \wedge E = F$,

(b) $E \wedge G = G \wedge E = G$.

In what follows we abbreviate $E \wedge F$ by EF.

```
for i := 0 to b − 1 do
    R = 1;
    for j := 0 to b − 1, j ≠ i do
        r(i, j) = ⋁_{(u,v)∈S_{f,i}×S_{f,j}} {x_k | u_k ≠ v_k}
    end
    R = R ∧ r(i, j);
    Express R in disjunctive normal form (DNF)
end
```

Algorithm 1: Algorithm to determine the collection of bases.

Example 2. Let $f : \{0, 1, 2, 3\}^5 \to \{1, 2, 3\}$ be a function whose values are included in Table 2.

Applying Algorithm 1 and the reduction rules of Lemma 2, we obtain

$$\begin{aligned} R &= (x_2 \vee x_5)(x_3)(x_1 \vee x_5)(x_1 \vee x_4) \\ &= x_1 x_2 x_3 \vee x_1 x_3 x_5 \vee x_3 x_4 x_5. \end{aligned} \tag{2}$$

Thus, the bases are $X_1 = \{x_1, x_2, x_3\}$, $X_2 = \{x_1, x_3, x_5\}$, and $X_3 = \{x_3, x_4, x_5\}$.

2.2. Interpolating Polynomials with No Redundant Variables. Given any partially defined function f over a finite field and any basis X for f, we would like to determine interpolating polynomials involving only those variables of X, that is, polynomials containing no redundant variables.

Definition 1. Let $f : D \to B$ where $D \subset A^n$ and let X be a basis for f. For all $c, d \in A^n$ define the relation $c \equiv_X d$ if and only if $c_i = d_i$, for each $x_i \in X$, $f(c)$, and $f(d)$ are defined, and $f(c) = f(d)$.

In other words, $c \equiv_X d$ if and only if $(\text{proj}_X c, f(c)) = (\text{proj}_X d, f(d))$. It is straightforward to verify the following.

Lemma 3. *$\equiv_X$ is an equivalence relation.*

Let $D/\equiv_X$ be a set of representatives of each equivalence class under $\equiv_X$.

Theorem 1. *Let F be a field, let (a_0, b_0), (a_1, b_1), …, (a_{k-1}, b_{k-1}) be a set of points in $F^n \times F$, and let X be a basis for the function f defined by $f(a_i) = b_i$, $i = 0, 1, \ldots, k-1$. Then an interpolating polynomial for f whose only variables are those of X is*

$$P(X) = \sum_{m=0,\ a'_m \in D/\equiv_X}^{k-1} b_m \prod_{j \neq m} \left(\frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} \right), \tag{3}$$

where ℓ_{jm} is the smallest index such that $x_{\ell_{jm}} \in X$ and a'_j and a'_m differ.

Proof. For each a_m, $m = 0, 1, \ldots, k-1$, let a'_m be the representative of the equivalence class C_m containing a_m. Then

$$\frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} = \begin{cases} 0 & \text{if } x = a_j \\ 1 & \text{if } x = a_m \end{cases} \tag{4}$$

and so $P(a_m) = b_m$, where b_m is the value of each tuple of C_m. □

Note that for each of the factors

$$c_{jm}(x) = \frac{x_{\ell_{jm}} - a_{j\ell_{jm}}}{a'_{m\ell_{jm}} - a_{j\ell_{jm}}} \tag{5}$$

in (3), $c_{jm}(a_j) = 0$ and $c_{jm}(a_m) = 1$ and so for all $i = 0, 1, \ldots, k-1$ and all $r = 1, 2, \ldots$

$$c^r_{jm}(a_i) = c_{jm}(a_i). \tag{6}$$

Equation (3) gives us a new algorithm for polynomial interpolation that involves only those variables in the chosen basis X, and furthermore, it also avoids redundant tuples by computing only on the representatives of the equivalence classes given by the relation $\equiv_X$.

Using (3), $O(k^2)$ operations are needed to interpolate k points for each function $f : F^n \to F$. Given k points of a function $f : F^n \to F^n$, the bases for the component functions f_i need not be the same. In this case, we can apply (3) n times, thus giving an algorithm with complexity $O(nk^2)$.

Example 3. Let us assume that the 0, 1, 2, 3 of Example 2 represent the corresponding elements of $\mathbb{Z}_5$. We found that the bases are $X_1 = \{x_1, x_2, x_3\}$, $X_2 = \{x_1, x_3, x_5\}$, and $X_3 = \{x_3, x_4, x_5\}$. Now let us use the new algorithm given by (3) to find an interpolating polynomial over $\mathbb{Z}_5$ in terms of the basis X_1. The equivalence classes under X_1 are

$$\begin{aligned} C_1 &= \{(0,2,1,2,3), (0,2,1,3,1)\}, \\ C_2 &= \{(2,1,1,0,1)\}, \\ C_3 &= \{(0,1,1,2,0)\}, \\ C_4 &= \{(2,1,2,1,0), (2,1,2,0,1), (2,1,2,2,1)\}, \\ C_5 &= \{(0,1,2,1,1)\}, \\ C_6 &= \{(2,2,2,2,2)\}. \end{aligned} \tag{7}$$

and the set of class representatives is

$$\begin{aligned} \frac{D}{\equiv_{X_1}} = \{&(0,2,1,2,3), (2,1,1,0,1), (0,1,1,2,0), \\ &(2,1,2,1,0), (0,1,2,1,1), (2,2,2,2,2)\}. \end{aligned} \tag{8}$$

Applying (3) and making use of (6), we have

$$\begin{aligned} P(X_1) = {} & 1 * \frac{x_1 - 2}{0 - 2}\frac{x_2 - 1}{2 - 1} \\ & + 1 * \frac{x_1 - 0}{2 - 0}\frac{x_3 - 2}{1 - 2}\frac{x_2 - 2}{1 - 2} \\ & + 2 * \frac{x_2 - 2}{1 - 2}\frac{x_1 - 2}{0 - 2}\frac{x_3 - 2}{1 - 2} \\ & + 2 * \frac{x_1 - 0}{2 - 0}\frac{x_3 - 1}{2 - 1}\frac{x_2 - 2}{1 - 2} \\ & + 3 * \frac{x_2 - 2}{1 - 2}\frac{x_1 - 2}{0 - 2}\frac{x_3 - 1}{2 - 1} \\ & + 3 * \frac{x_1 - 0}{2 - 0}\frac{x_2 - 1}{2 - 1} \\ = {} & 2(x_1 - 2)(x_2 - 1) + 3x_1(x_2 - 2)(x_3 - 2) \\ & + 4(x_1 - 2)(x_2 - 2)(x_3 - 2) \\ & + 4x_1(x_3 - 1)(x_2 - 2) \\ & + 4(x_1 - 2)(x_2 - 2)(x_3 - 1) + 4x_1(x_2 - 1). \end{aligned} \tag{9}$$

Thus, a polynomial which depends only on the variables x_1, x_2, and x_3 and interpolates the tuples given by Example 2 is

$$P(X_1) = 4x_1x_2 + 4x_2x_3 + 3x_1 + 2x_3 + 1. \tag{10}$$

3. Data Representation

As can be deduced from the previous section, an implementation of the multivariate polynomial interpolation algorithm must be able to represent and compute on disjunctive normal form (DNF), conjunctive normal form (CNF), and multivariate polynomial expressions. The following subsections establish our method of representation, which ultimately determines the techniques used for implementing the algorithms.

3.1. DNF and CNF Expressions. A Boolean expression D in DNF using only uncomplemented variables is a disjunction of conjunctive clauses $D = \bigvee_{i=0}^{q-1} c_i$, where $c_i = x_0^{t_{i_0}} \wedge x_1^{t_{i_1}} \wedge \cdots \wedge x_{m-1}^{t_{i_{m-1}}}$, $t_{i_j} \in \{0, 1\}$. We shall refer to each c_i term as a *cube*. For example, the following is a Boolean expression in DNF: $x_1x_2 \vee x_1x_3 \vee x_0$. In our scheme, a DNF expression is represented as a set of cubes $\{c_0, c_1, \ldots, c_{q-1}\}$ where each c_i is represented as a sequence $\langle t_{i_0}, t_{i_1}, \ldots, t_{i_{m-1}} \rangle$. Observe that complemented variables are not used in the algorithm for finding essential variables and bases, hence their representation is not necessary in this context.

Example 4. The DNF expression $D(x_0, x_1, x_2, x_3) = x_1x_2 \vee x_1x_3 \vee x_0$ is represented as $\{\langle 0, 1, 1, 0\rangle, \langle 0, 1, 0, 1\rangle, \langle 1, 0, 0, 0\rangle\}$

A Boolean expression C in CNF using only uncomplemented variables is represented as a set of disjunctions of literals (DOL) $\{d_0, d_1, \ldots, d_{q-1}\}$ where each DOL d_i

is represented as a sequence $\langle t_{i_0}, t_{i_1}, \ldots, t_{i_{m-1}} \rangle$. Within our implementation there is no need to distinguish the CNF from the DNF representation since no individual component needs to operate on both simultaneously.

Example 5. The CNF expression $C(x_0, x_1, x_2) = (x_0 \vee x_2) \wedge (x_0 \vee x_1) \wedge (x_1)$ is represented as $\{\langle 1, 0, 1\rangle, \langle 1, 1, 0\rangle, \langle 0, 1, 0\rangle\}$

3.2. Polynomial Representation. A multivariate polynomial P over $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime is defined as $P(x_0, x_1, \ldots, x_{n-1}) = \sum_{i=0}^{q-1} m_i$, where $m_i = \alpha_i x_0^{t_{i_0}} x_1^{t_{i_1}} \cdots x_{n-1}^{t_{i_{n-1}}}$ and $\alpha_i, t_{i_j} \in \mathbb{Z}_p$. For example, one multivariate polynomial over $\mathbb{Z}_5$ is $P(x_0, x_1, x_2) = 4 + x_0^3 x_1 + 3x_2 + x_2^4$. In our scheme, a polynomial is represented as a set of monomials $\{m_0, m_1, \ldots, m_{q-1}\}$ where each m_i is represented as a tuple $(\alpha_i, \langle t_{i_0}, t_{i_1}, \ldots, t_{i_{n-1}}\rangle)$. Binary representation requires $\lceil \log_2 p \rceil$ bits for each α_i and t_{i_j}.

Example 6. The multivariate polynomial $P(x_0, x_1, x_2) = 4 + x_0^3 x_1 + 3x_2 + x_2^4$ over $\mathbb{Z}_5$ is represented as $\{(4, \langle 0, 0, 0\rangle), (1, \langle 3, 1, 0\rangle), (3, \langle 0, 0, 1\rangle), (1, \langle 0, 0, 4\rangle)\}$.
For the three variable polynomial over $\mathbb{Z}_5$ each monomial requires $4 \times \lceil \log_2 5 \rceil = 12$ bits for binary representation.

4. General Description of Implementation

Our algorithm for determining interpolating polynomials that do not contain any redundant variables consists of two stages. The first stage identifies the dependent variables using Sasao's algorithm, while the second uses this information along with (3) to determine P_i. Thus, a key part of the reconfigurable logic implementation is choosing efficient hardware structures to perform the critical parts of both stages. For the first stage, the most time-consuming task is the conversion of a Boolean expression in conjunctive normal form (CNF) to disjunctive normal form (DNF). The second step relies heavily on the identification of distinct binomials as they are generated from (3) and multivariate polynomial multiplication/addition.

To take advantage of the FPGA's fine-grained parallelism and considering their limitations in I/O bandwidth, these computations were implemented in a pipelined manner. In other words, computational blocks were designed to sustain stream processing as allowed by the FPGA's resources, rather than accumulate all needed data and then perform block processing.

Figure 1 shows a block diagram of the implementation. The *CNF Generator* receives the n-tuples and performs Algorithm 1 to generate the disjuntions of literals, that is, the $r(i, j)$ terms. These are streamed to the *Cover DOLs* block which eliminates redundant disjuntions of literals by using Lemma 2. The nonredundant DOLs are then fed to block *CNF2DNF* which computes the conjunctions of literals and uses Lemma 2 to eliminate redundant conjunctions. The output of *CNF2DNF* is the set of bases. A priori biological knowledge about the organism under study is required to choose the most adequate base. The n-tuples are also fed to the second stage which uses the chosen base to determine representatives of the equivalence classes from the n-tuples. The representatives are then streamed to the modified *Lagrange* interpolation block which implements (3) by generating the binomials for each pair of class representatives (*Binomial Generator*), eliminates duplicate binomials (*Filter Duplicates*), then multiplies and adds the product results.

The highlighted blocks in Figure 1, that is, *Cover DOLs*, *Cover COLs*, *Determine Class Representatives*, and *Polynomial Addition*, have two characteristics that make them suitable candidates for systolic array processing: (1) each performs an operation on a sequence of elements that are streamed from the preceeding block, (2) the operation is such that it performs reduction of a sequence of elements from multivariate polynomials or Boolean expressions. For example, the *Cover Cubes* receives preliminary results from the CNF to DNF conversion and eliminates redundant cubes using Lemma 2. This is an operation that can be described as given a sequence C of cubes $(C = c_1, c_2, \ldots, c_m)$ determine a subsequence $C' \subset C$ such that it contains only cubes $c_i \in C$ where there exists no other $c_j \neq c_i \in C$ such that $c_i \wedge c_j = c_j$.

The following sections describe a generalized algorithm and systolic array-based architecture for the type of reduction operations common throughout our interpolation methodology. This is followed by a description of how they are utilized as part of the architectural data path.

5. Generalization of Reduction Tasks

The subtasks that deal with reduction in our interpolation algorithm can be generally described as follows. Given the sequence $X = x_1, x_2, \ldots, x_m$ of elements from Σ, compute the sequence $Y = y_1, y_2, \ldots, y_m$ where $y_i \in \Sigma_\varepsilon = \Sigma \cup \varepsilon$ by eliminating or combining elements based on binary comparisons between them. The empty element (ε) is introduced for the representation of operations where groups of elements can be reduced to a single element. Algorithm 2 shows a general form of these problems, where W and S are binary functions of the form $\Sigma_\varepsilon \times \Sigma_\varepsilon \rightarrow \Sigma_\varepsilon$, and the symbol $\|$ denotes parallel computation.

The tasks of distinctness, polynomial addition, and redundant Boolean term elimination (hereon referred to as *cover*) can be implemented by specifying W and S, as explained in the following subsections.

5.1. Distinctness. The task of identifying the distinct elements of a sequence $X = x_1, x_2, \ldots, x_m$ is needed within our interpolation algorithm to determine class representatives and filter out binomial duplicates. It can be implemented by Algorithm 2 with

$$W\left(x_i, x_j\right) := \begin{cases} x_j & \text{if } x_i = \varepsilon \\ x_i, & \text{otherwise,} \end{cases}$$
$$S\left(x_i, x_j\right) := \begin{cases} \varepsilon & \text{if } x_i = x_j \quad \text{or} \quad x_i = \varepsilon \\ x_j, & \text{otherwise.} \end{cases} \tag{11}$$

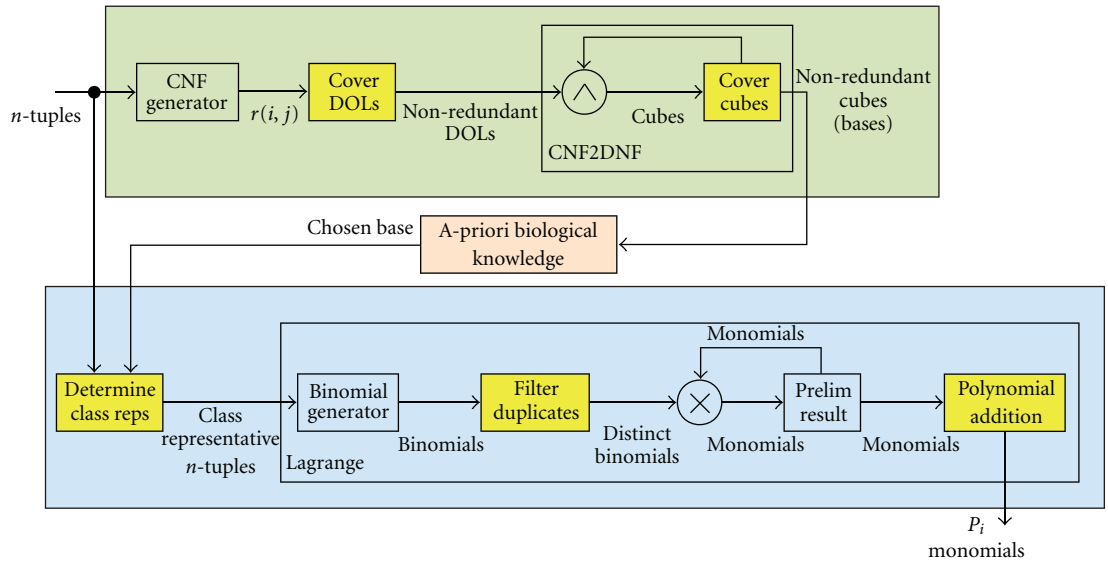

Figure 1: Block diagram of the implementation, highlighting the subtasks that are implemented using the systolic array architecture.

```
for i := 1 to n − 1 do
    for j := i + 1 to n do
        x_i := W(x_i, x_j) ‖ x_j := S(x_i, x_j);
    end
end
```

Algorithm 2: Nested loop algorithm for reduction tasks.

Example 7. Assume $X = 5, 8, 8, 5, 8, 1$. After computation with Algorithm 2 using (11), the result is $X' = 5, 8, 1, \varepsilon, \varepsilon, \varepsilon, \varepsilon$.

We provide a correctness proof for this operation. Proofs of the other reduction operations given in this section are similar.

Lemma 4. *For an input sequence $x = x_1, x_2, \ldots, x_m$, where $x_i \in \Sigma$, the output of Algorithm 2 with functions W and S defined as in (11) outputs $x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \varepsilon, \varepsilon, \ldots, \varepsilon$ where $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is the subsequence of distinct elements of the input sequence.*

Proof. We show by induction that after $j \leq k$ iterations of the outer loop,

$$x_i, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_j}, y_{j+1}, \ldots, y_m, \tag{12}$$

where for each $p \geq j+1, y_p \in \{x_{i_{j+1}}, \ldots, x_{i_k}\} \cup \{\varepsilon\}$.

After $j = 1$ iterations, $x_1 = x_{i_1}$. Next suppose that after j iterations,

$$x_1, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, \ldots, x_{i_j}, y_{j+1}, \ldots, y_m. \tag{13}$$

for some $j < k$ where $y_p \in \{x_{i_{j+1}}, \ldots, x_{i_k}\} \cup \{\varepsilon\}$ for each $p \geq j+1$. Then either (1) $y_{j+1} = x_{j+1}$ or (2) $y_{j+1} = \varepsilon$. In the first case, $y_{j+1} = x_{i_{j+1}}$ since x_{j+1} is not equal to any $x_p, p \leq j$ and is not changed by the $j+1$st iteration. In the second case, $y_{j+1} = \varepsilon$ will be replaced by $x_{i_{j+1}}$ during the $j+1$st iteration. Hence in either case,

$$x_1, x_2, \ldots, x_j, x_{j+1}, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_{j+1}}, y_{j+2}, \ldots, y_m, \tag{14}$$

where for each $p \geq j+2, y_p$ is either ε or an element x_{i_p}, where $p \geq j+2$.

Hence after $i = k$ iterations,

$$x_1, x_2, \ldots, x_m = x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \varepsilon, \ldots, \varepsilon \tag{15}$$

and remains the same after $i > k$ iterations since $W(\varepsilon, \varepsilon) = S(\varepsilon, \varepsilon) = \varepsilon$. □

5.2. Polynomial Addition. Assume that each of the elements in $X = x_1, x_2, \ldots, x_m$ is a monomial. A polynomial addition operation can be implemented by Algorithm 2 with

$$W\left(x_i, x_j\right) := \begin{cases} x_i + x_j & \text{if } \operatorname{mon}(x_i) = \operatorname{mon}\left(x_j\right), \\ & \text{or } \quad x_i = \varepsilon, \\ x_i, & \text{otherwise}, \end{cases}$$
$$S\left(x_i, x_j\right) := \begin{cases} \varepsilon & \text{if } \operatorname{mon}(x_i) = \operatorname{mon}\left(x_j\right), \\ & \text{or } \quad x_i = \varepsilon, \\ x_j, & \text{otherwise}, \end{cases} \tag{16}$$

where $\operatorname{mon}(x_i) = \langle t_{i_0}, t_{i_1}, \ldots, t_{i_{n-1}} \rangle$, that is, the monomial without the coefficient.

Example 8. Assume $X = x^2, xy^2, 3x^2, xy, 2xy$. After computation with Algorithm 2 using (16), the result is $X' = 4x^2, xy^2, 3xy, \varepsilon, \varepsilon$.

5.3. Cover. Assume that each of the elements in X is a cube. A cube c_i covers another cube c_j (represented as $c_i \supset c_j$) if all Boolean variables present in c_i are present in c_j (e.g., $x_2 \supset x_1x_2x_3$, but $x_2x_4 \not\supset x_1x_2x_3$). The cover operation can be implemented by Algorithm 2 with

$$W\left(x_i, x_j\right) := \begin{cases} x_j & \text{if } x_j \supset x_i, \\ x_i, & \text{otherwise,} \end{cases}$$
$$S\left(x_i, x_j\right) := \begin{cases} \varepsilon & \text{if } x_i \supset x_j, \\ x_j, & \text{otherwise,} \end{cases} \tag{17}$$

where $q_i \supset \varepsilon$ for any q_i.

Example 9. Assume $X = x_1x_2, x_2x_4, x_1, x_1x_2x_4$. After computation with Algorithm 2 using (17), the result is $X' = x_1, x_2x_4, x_1, \varepsilon$. Notice that there may be duplicate terms in the result but these can be easily eliminated using a distinctness operation.

Algorithm 2 is also amenable to other problems that can be expressed as the result of binary comparisons between every two elements of a sequence. For example, the problem of sorting a sequence S can be implemented with

$$W\left(x_i, x_j\right) := \min\left(x_i, x_j\right),$$
$$S\left(x_i, x_j\right) := \max\left(x_i, x_j\right). \tag{18}$$

6. Hardware Structure for Reduction Tasks

This section explains our proposed hardware structure and outlines the mapping for the reduction tasks. We first discuss the systolic array and cells with the assumption that the array is deep enough to process the input sequence without overflowing. We proceed by discussing the additional components that must be added to guarantee correct processing even if the overflow condition does not hold.

6.1. Systolic Array and Cells. The proposed structure is a linear systolic array of n computational cells, each performing an operation deduced from W and S of the above discussion. Figure 2 illustrates the array and contents of the basic cell. Each cell contains two registers F and M, where each has enough resources to hold any single element from the alphabet Σ_ε. Each cell also contains the logic to perform functions $W(F, M)$ and $S(F, M)$. All cells are initialized to the empty symbol. At each clock step a new element from the sequence X is input to the first cell and every cell performs a comparison between its stored value and the value from the previous cell and assigns (possibly) new values to the registers according to W and S. After all elements have been processed the M registers contain the resulting sequence and can be shifted out of the array using the connections between the M registers.

More formally, our linear systolic array computation can be modeled as a simplified version of the generic systolic array presented in [16], as shown in Algorithm 3. The notation $F_i[r]$ symbolizes the content of register F in cell r at iteration i.

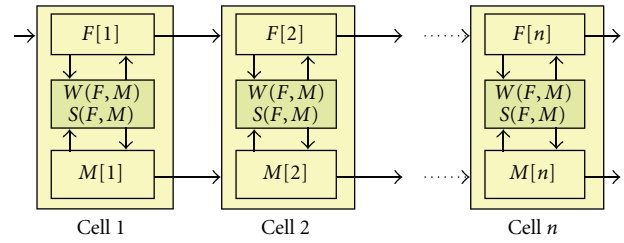

Figure 2: Block diagram of systolic array.

$i := 0$
$\|_{r=1}^{n} \begin{cases} M_0[r] := \varepsilon \\ F_0[r] := \varepsilon \end{cases}$
while stop_condition = false **do**
  $i := i + 1$
  if $i < m$ **then** $F_{i-1}[0] := x_i$
  else $F_{i-1}[0] := \varepsilon$
  end if
  $\|_{r=1}^{n} \begin{cases} F_i[r] := S(F_{i-1}[r-1], M_{i-1}[r]) \\ M_i[r] := W(F_{i-1}[r-1], M_{i-1}[r]) \end{cases}$
end

Algorithm 3: Model of the linear systolic array computation.

Table 3: Illustration of index generation by Algorithm 4 for the example shown in Figure 3.

$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$	$j = 8$	$j = 9$
x_0, x_1	x_0, x_2	x_0, x_3	x_0, x_4	x_0, x_5	x_1, x_5	x_2, x_5	x_3, x_5	x_4, x_5
		x_1, x_2	x_1, x_3	x_1, x_4	x_2, x_4	x_3, x_4		
				x_2, x_3				

Figure 3 illustrates the operation of the array using cells that implement the distinctness task by defining W and S as in (11). Observe that the operations implied by W and S and the connections between neighboring cells accomplish the comparisons established in Algorithm 2, albeit in a different order and in a parallel manner. In fact, the computation implemented by the systolic array can be interpreted as a reindexing of Algorithm 2 as presented in Algorithm 4. Table 3 shows the comparison indexes generated by Algorithm 4 for the example in Figure 3. Notice that even though Algorithm 4 has reordered the indexes, the same data dependence is maintained as in Algorithm 2. In other words, if $f(x_i, x_j)$ is performed before a $f(x_i, x_k)$ (where $j \neq k$) in Algorithm 2 the same order is preserved in Algorithm 4. The same condition holds for the order of $f(x_i, x_j)$ and $f(x_k, x_j)$.

6.2. Processing Sequences Longer than the Array Depth. The functionality of the basic cells mandates their implementation on user logic inside the FPGA (rather than on embedded

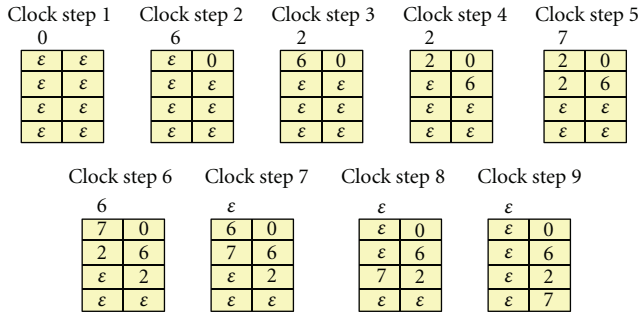

Figure 3: Systolic array implementing the distinctness operation for $X = 0, 6, 2, 2, 7, 6$. Each row represents a cell. Within each cell, the left and right columns represent the S and M registers, respectively. Elements are input to the top row and flow downwards.

```
for j := 1 to 2n − 3 do
    parallel for i := iif (j ≤ n, 1, j − n) to |(j)/2| do
        x_i = W(x_i, x_{j−i+1}) ‖ x_{j−i+1} = S(x_i, x_{j−i+1})
    end
End
```

Algorithm 4: Reindexed algorithm for reduction tasks.

units, such as block RAMS). Depending on the application, the characteristics of the data sequence, and FPGA model, the FPGA might not have enough resources to instantiate a systolic array whose depth d_{sa} is greater than or equal to m (the sequence length). In such cases, there is no guarantee that the array by itself will be able to compare each element against each other to obtain the correct solution. To illustrate this anomaly, assume that in the example presented in Figure 3 the input is a sequence of *distinct* elements $X = x_0, x_1, \ldots, x_5$. The array would be filled once x_3 is registered in the last cell, hence none of the cells would have operated on the combination of x_4 and x_5. A control and temporary storage mechanism must be provided to reiterate the data through the pipeline if needed and guarantee that all elements are compared against each other. This can be accomplished by a scheme such as the one shown in Figure 4 through the addition of a FIFO, control unit and several datapath control elements (i.e., multiplexers).

The *Overflow FIFO* (OF) stores elements that make it through the array without having been eliminated or registered. These elements will have to be reiterated through the array to test their relation to other elements in the OF. For example, suppose we have an array of depth $d_{sa} = 4$ that implements the distinctness operation. The sequence $\langle 3, 5, 1, 3, 2, 6, 7, 6\rangle$ will fill the pipeline with $\langle 3, 5, 1, 2\rangle$ so that the final $\langle 6, 7, 6\rangle$ overflows the pipeline. These 3 elements are stored in the OF for a later pass through the array. The FIFO depth $d_f \geq m - d_{sa}$ where m is the number of elements in the sequence and d_{sa} is the array depth. Although both systolic array cells and FIFO spend mostly on register resources, a

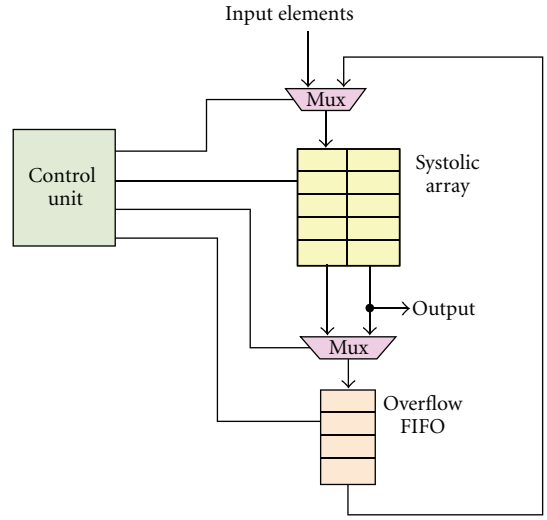

Figure 4: Pipeline with added overflow FIFO and control unit to support sequences longer than n.

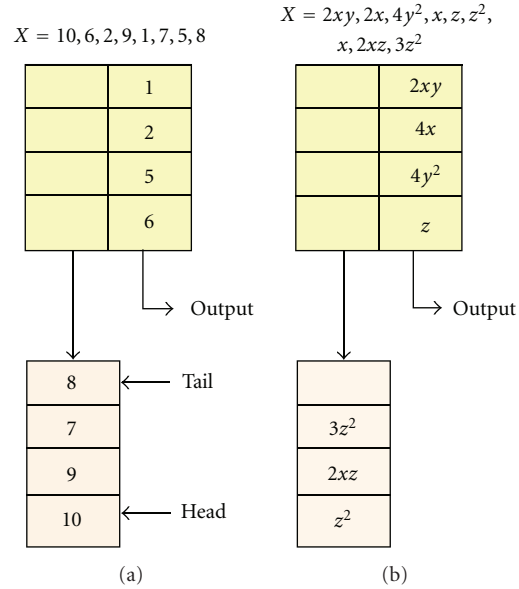

Figure 5: Contents of the systolic array and OF after a first pass of the sequences through (a) sorting and (b) polynomial addition.

reason why one may be able to increase FIFO depth and not necessarily array depth is that in FPGAs FIFOs are easily mapped to the embedded block RAM, whereas processing cells map to user logic.

The *control unit* controls the data flow between the array and the OF and determines how to reiterate the overflowed data through the array. For the reduction tasks presented in Section 5 we can deduce three basic modes of operation for the control unit. Figures 5 and 6 illustrate the need for these modes. Figure 5 shows the end of a first pass of a sequence through systolic arrays that implement (a) a sorting algorithm, and (b) sum of polynomials.

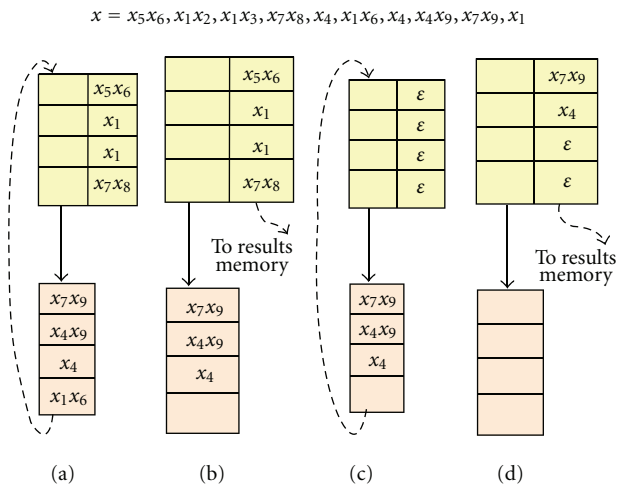

Figure 6: Contents of systolic array and overflow FIFO for the shown sequence X (a) after first pass, (b) after second pass (OF to array refeed), (c) systolic array is emptied for new OF refeed, (d) after final refeed from OF. Dotted lines illustrate the paths of data refeed.

(i) In the case of a nonreducing operation like sorting (Figure 5(a)), the array will contain d_{sa} sorted elements. Thus, elements in the array can be shifted out while elements in the OF are reinputted into the array. This must be repeated until the OF is empty after a pass, that is, at most $\lceil m/d_{sa} \rceil$ times.

(ii) When adding polynomials (Figure 5(b)), the array will contain at most d_{sa} monomials with their final coefficients. Thus, elements in the array can be shifted out while elements in the OF are reinputted into the array. This must be repeated until the OF is empty after a pass, that is, at most $\lceil m/d_{sa} \rceil$ times.

Figure 6 illustrates the various passes needed for the cover operation. At the end of a first pass the array will contain at most d_{sa} cubes. However, in this case a first pass through the array does not guarantee that all cubes in the OF have been compared to all the cubes registered in the array. For example, in Figure 6(a) cube x_1x_6 passed through the array without being registered. Meanwhile, the last cube x_1 covered x_1x_3 and x_1x_2 in two cells of the array. Thus, we have at least one cube (x_1x_6) in the OF which was not compared to a cube in the array (x_1). The elements in the OF must be refed through the (unflushed) array to allow every c_i stored in the FIFO to be compared against every c_j registered in the array. The refeed is repeated until a complete pass of the OF cubes through the array does not modify the cubes registered in the array. After this (Figure 6(b)), if any c_i in the OF is going to be part of the result, for example, x_4, it must only be still compared to other elements in the OF. The contents of the array may be shifted out to a memory that stores the results, the array is reset and the OF cubes are refed (Figure 6(c)). The three-step process continues until the OF is empty after an OF refeed.

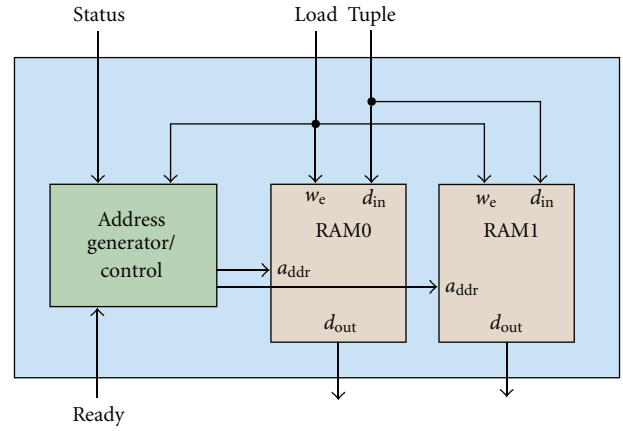

Figure 7: Element pair generator.

7. Further Implementation Considerations

Aside from the reduction tasks, the computational blocks in our implementation can be classified as performing one of two functionalities: element pair generation or multiple term multiplication. Element pair generators, such as the *CNF Generator* and the *Binomial Generator* in Figure 1, input a set of tuples and output a distinct pair of tuples at each computational step. This is accomplished by using the architecture illustrated in Figure 7. As tuples are input (each accompanied by a load signal) both RAMs store the tuples and an Address Control Generation unit counts the number of tuples. When a status signal indicates that all tuples have been received, the ACG begins generating all possible address combinations (a_0, a_1) for $0 \le a_0 < n$, $a_0 + 1 \le a_1 \le n$. The distinct pairs of n-tuples produced by the *CNF Generator* are fed to $n\lceil \log_2 p \rceil$-comparators to compute the $r(i, j)$ terms in Algorithm 1, as shown Figure 8. In the case of the *Binomial Generator*, the corresponding terms within the distinct pairs of n-tuples are subtracted from each other and the results are used to determine the terms of the binomial for (3), as shown in Figure 9.

Multiple-term multiplication/conjunction, such as needed in blocks *CNF2DNF* and *Lagrange*, is performed using the architectures shown in Figures 10 and 11, respectively. The architecture depicted in Figure 10 receives DOLs from the *CNF Generator*, the *DOL2Literals* block generates a literal expression for each found in the DOL. The literals are queued in a circular FIFO and are conjuncted to each of the results in the *Preliminary Results FIFO*. After each DOL is conjuncted, the resulting cubes are passed through the *Cover Cubes* block which eliminates redundant cubes. A control unit monitors and generates signals to coordinate the data path activities.

The architecture depicted in Figure 11 performs the computations of (3). Each binomial that is received from the *Binomial Generator* is registered. Then each of its two monomials is multiplied by each of the monomials that has resulted from previous binomial multiplications. The monomials that result from the multiplication are input to the *PrelimPolyAdd* block which adds similar monomials to

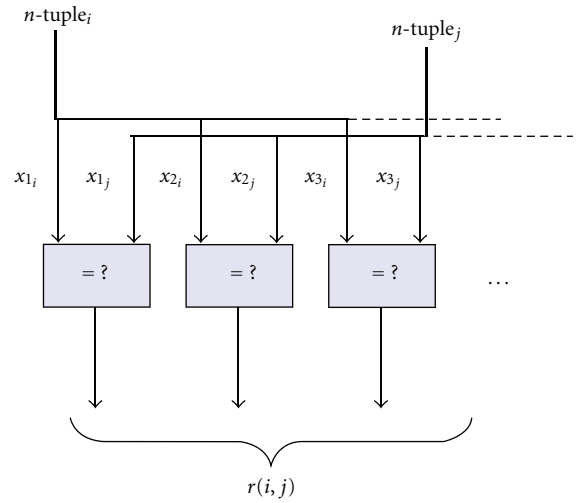

Figure 8: Computation of the disjunction of literals $r(i, j)$ from a pair of n-tuples.

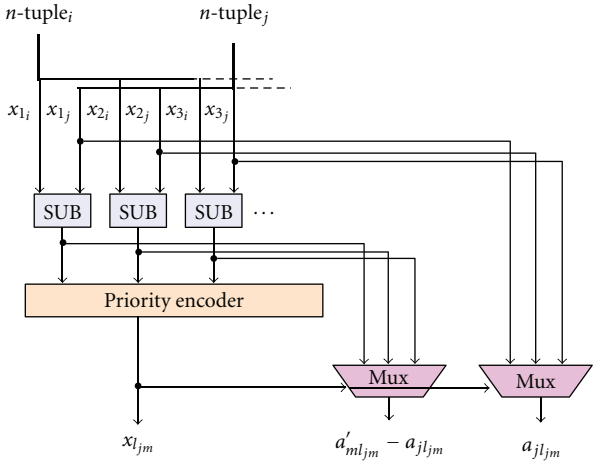

Figure 9: Computation of the terms of a binomial for (3) from a pair of n-tuples.

reduce the size of the preliminary product. *PrelimPolyAdd* outputs its result to the *PrelimRes FIFO*. The iterative process continues until the last binomial of a product is received, in which case each monomial of the preliminary product is output to the final *Polynomial Addition* block (see Figure 1). Once the last binomial of the last product has been multiplied, the (final) *Polynomial Addition* block is signaled and begins to output the monomials of the final result.

8. Results and Discussion

This section presents and discusses results for the individual reduction tasks as well as the complete interpolation algorithm implementation.

8.1. Reduction Tasks. The systolic array cells for the subtasks of distinctness, polynomial addition, and Boolean cover were modeled using Verilog HDL, based on their function

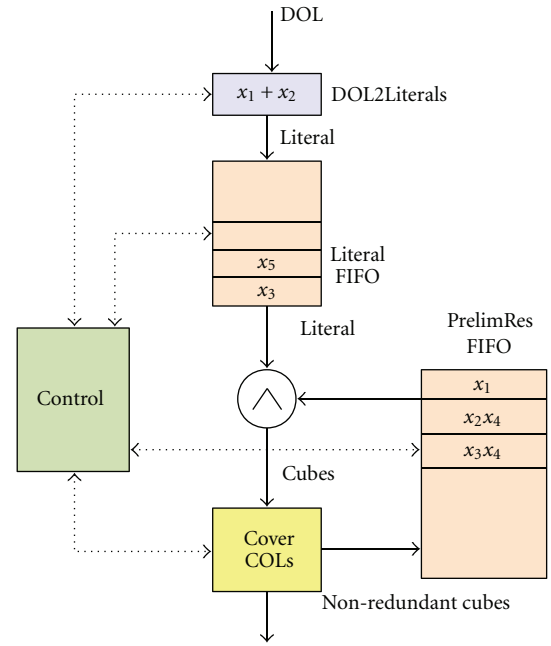

Figure 10: Illustration of a step in the conversion of CNF $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_3 \vee x_5) \wedge (x_1 \vee x_2)$ to DNF. The first two conjunctions have been computed and their redundant terms eliminated and stored in the Preliminary Results FIFO.

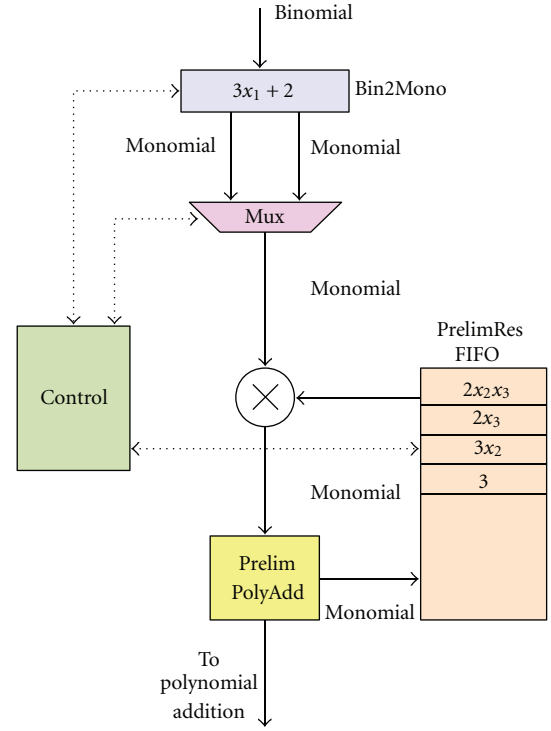

Figure 11: Illustration of a step in the multiplication of binomials $(x_2 + 1)(2x_3 + 3)(3x_1 + 2)$ in $\mathbb{Z}_5$. The multiplication of the first two binomials has completed and the results are stored in the *Preliminary Results FIFO*.

Table 4: Experimental results for subtask blocks.

	d_{sa}	Slices*	Slice F/F*	4 input LUTs*	FIFO16/RAMB16*	freq (Mhz)	t_F (s)	t_C (s)	speedup
3∗Cover	64	4141 (4.65%)	4348 (2.44%)	7708 (4.33%)	9 (2.68%)	176	$1.12E-03$	$1.29E-02$	11.51×
	128	8279 (9.29%)	8700 (4.88%)	15054 (8.45%)	9 (2.68%)	176	$5.90E-04$	$1.29E-02$	21.93×
	256	16554 (18.58%)	17404 (9.77%)	30796 (17.28%)	9 (2.68%)	176	$3.21E-04$	$1.29E-02$	40.33×
3∗P-add	64	2967 (3.33%)	3516 (1.97%)	4119 (2.31%)	7 (2.08%)	200	$6.84E-04$	$3.36E-02$	49.17×
	128	5885 (6.61%)	7455 (4.18%)	7156 (4.02%)	7 (2.08%)	200	$3.58E-04$	$3.36E-02$	93.99×
	256	11741 (13.18%)	14880 (8.35%)	14228 (7.99%)	7 (2.08%)	200	$1.95E-04$	$3.36E-02$	172.28×
3∗Distinct	64	2680 (3.01%)	3741 (2.10%)	3173 (1.78%)	7 (2.08%)	190	$7.20E-04$	$3.37E-02$	46.87×
	128	5342 (6.00%)	7465 (4.19%)	6261 (3.51%)	7 (2.08%)	190	$3.77E-04$	$3.37E-02$	89.60×
	256	10659 (11.96%)	14903 (8.36%)	12437 (6.98%)	7 (2.08%)	190	$2.05E-04$	$3.37E-02$	164.25×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

definitions presented in Section 5. The array and control units were also modeled using Verilog based on the descriptions provided in Section 6. Cell, systolic array and control unit models were designed as parameterizable modules in terms of width of elements, array depth and control mode, respectively. Models were behaviorally simulated using ModelSim SE 6.0 to validate their results against software versions of the algorithms. Xilinx ISE Design Suite 11.1 was used to synthesize the architectures for a Virtex-4 XC4VLX200ff1513-11 in the DRC Dev Systems DS2000. Default synthesis parameters were used, such as mapping FIFOs to block RAMs and speed as the optimization goal. All the reported results for the individual tasks are before Place and Route.

For each of the three subtasks, several systolic array depths (d_{sa}) were implemented, then randomly generated sequences of various bit widths and lengths were input. Table 2 shows timing and resource results for (1) the cover operation on sequences of length 4096 and width of 32 bits, (2) the polynomial addition operation on sequences of length 4096, where each element represents a monomial of 8 variables in $\mathbb{Z}_5$, and (3) the distinctness operation on the same sequences as (3). Results are compared against a software implementation compiled using the GNU project c++ compiler with optimization level 3 and running on a 3.40 GHz Intel Pentium D CPU with 2 MB cache and 3 GB RAM. CPU execution times were measured by accessing the processor's cycle counter. In all cases, we report the smallest measured CPU execution time. The *freq* column indicates the FPGA clock frequency, t_F and t_C are the FPGA and CPU run times, and speedup is computed as t_C/t_F. The numbers shown in parenthesis next to each resource quantity are the resource percentages for the target device.

Several important observations can be highlighted from the results in Table 4. First, the maximum operation frequency is independent of the array depth, allowing performance to scale adequately with depth. In fact, given the simplicity of the required control unit and storage, the longest path was mostly determined by the implementation of functions W and S inside the basic cells. Second, the parallelism achieved through the use of the systolic array greatly compensates for the refeeds that must be performed when the result is longer than the array depth. For instance, the result for the cover operation was of length 3386 on average, yet there is a speedup of more than 40× for $d_{sa} = 256$. Finally, the resource utilization to achieve competitive speedups is minimal and easy to estimate when scaling since it is almost completely attributable to the systolic array cells and overflow FIFO. Hence when using this structure as part of a bigger design, the designer could readily determine which parameters are best suited for the area/performance constraints and objectives.

Part of our purpose here is to argue that the reduction tasks can be implemented efficiently using the systolic array design. With Table 4, we intend to demonstrate the effect of each particular operation for users that might want to use any of them as part of a bigger design. Clearly, the ultimate speedup may be dictated by other components of the final design, as shown by Tables 5 and 6.

8.2. Interpolation Algorithm. We modeled the proposed interpolation algorithm in Verilog HDL using the reduction tasks and the architectures described in previous sections as building blocks. The behavioral simulation and synthesis tools as well as the software compilation parameters and platform were the same as in the reduction task experiments.

The results are shown separately for each of the two interpolation stages since, once the bases have been computed by the first stage, a scientist's intervention would be necessary to choose the most appropriate before proceeding to the second stage. For both stages, the synthesis tool determined maximum operating frequencies above 150 MHz, thus we chose 150 MHz for the experiments. The Xilinx ISE 11.1 place and route tools (with default effort level) were able to meet the timing requirements reported for Stages 1 and 2. We attribute this to the fact that the great majority of connections in our design are local and grow only linearly with n.

8.2.1. Stage 1. Table 5 reports timing and resource results for the first stage. Randomly generated sets of 100 n-tuples ($8 \leq n \leq 13$) were input to implementations with diverse systolic array depths. The table shows results for the d_{SA} case with the shortest execution time for each n. The smaller values of n generated fewer intermediate terms during computation,

Table 5: Experimental results for stage 1.

n	d_{SA}	Slices	Slice FFs*	4-input LUTS*	FIFO16/ RAMB16*	t_F (ms)	t_C (ms)	Speedup
8	32	1651 (1.85%)	1908 (1.07%)	3047 (1.71%)	25 (7.44%)	0.04	1.27	29.59×
9	64	3011 (3.38%)	3596 (2.02%)	5515 (3.1%)	27 (8.04%)	0.19	2.64	13.87×
10	128	5860 (6.58%)	7201 (4.04%)	11186 (6.28%)	33 (9.82%)	0.49	7.22	14.78×
11	128	6191 (6.95%)	7728 (4.34%)	11755 (6.60%)	43 (12.80%)	1.01	22.97	22.74×
12	256	11880 (13.34%)	14907 (8.37%)	22573 (12.67%)	55 (16.37%)	2.54	82.28	32.46×
13	256	14101 (15.83%)	16981 (9.53%)	25676 (14.41%)	95 (28.27%)	8.89	596.9	67.13×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

Table 6: Experimental results for stage 2.

n	Basis Vars	Class Reps	Slices*	Slice FFs*	4-input LUTs*	FIFO16/ RAMB16*	t_F (ms)	t_C (ms)	Speedup
8	2	23	7745 (8.69%)	8335 (4.68%)	13846 (7.77%)	28 (8.33%)	0.10	1.59	16.67×
8	3	63	7950 (8.92%)	8352 (4.69%)	14205 (7.97%)	28 (8.33%)	0.44	7.76	17.68×
8	4	90	7955 (8.93%)	8359 (4.69%)	14213 (7.98%)	30 (8.93%)	0.84	16.60	19.71×
9	2	25	8520 (9.56%)	9220 (5.17%)	15205 (8.53%)	28 (8.33%)	0.10	1.85	17.64×
9	3	77	8539 (9.58%)	9226 (5.18%)	15239 (8.55%)	28 (8.33%)	0.69	11.20	16.13×
9	4	94	8582 (9.63%)	9260 (5.2%)	15340 (8.61%)	34 (10.12%)	1.14	20.26	17.81×
10	2	24	9188 (10.31%)	10098 (5.67%)	16710 (9.38%)	31 (9.23%)	0.10	1.78	17.95×
10	3	65	9192 (10.32%)	10106 (5.67%)	16714 (9.38%)	31 (9.23%)	0.47	8.30	17.69×
10	4	92	9201 (10.33%)	10130 (5.69%)	16742 (9.40%)	33 (9.82%)	0.86	18.43	21.36×
11	2	25	10130 (11.37%)	10978 (6.16%)	18024 (10.12%)	34 (10.12%)	0.11	2.01	18.80×
11	3	75	10156 (11.4%)	11001 (6.17%)	18067 (10.14%)	36 (10.71%)	0.58	12.80	22.22×
11	4	92	10115 (11.35%)	11008 (6.18%)	18025 (10.12%)	37 (11.01%)	1.05	23.00	21.93×
12	2	25	10861 (12.19%)	11861 (6.66%)	19077 (10.71%)	36 (10.71%)	0.10	2.05	19.93×
12	3	70	10806 (12.13%)	11881 (6.67%)	19012 (10.67%)	38 (11.31%)	0.52	10.68	20.41×
12	4	93	10877 (12.21%)	11895 (6.68%)	19121 (10.73%)	38 (11.31%)	1.03	19.90	19.26×
13	2	25	11405 (12.8%)	12747 (7.15%)	20240 (11.36%)	39 (11.61%)	0.10	2.13	20.74×
13	3	71	11469 (12.87%)	12760 (7.16%)	20358 (11.43%)	40 (11.90%)	0.53	12.50	23.68×
13	4	91	11416 (12.81%)	12771 (7.17%)	20274 (11.38%)	41 (12.20%)	0.85	21.38	25.21×

*Resource percentages are for a Virtex-4 XC4VLX200ff1513-11.

hence smaller values of d_{SA} produced better results since they incurred in less unnecessary overhead for filling and emptying. As the number of variables increases, the number of terms produced by Algorithm 1 increases exponentially, which is clearly evidenced by the run times, that is, columns t_F and t_C. The FPGA implementation achieves speedups of 13× to 67×, which mostly increase with n. The increase in speedup at the higher values of n can be attributed to a higher percentage of time of full pipeline utilization as well as the increased benefit of parallelism in the comparisons between the n-tuple variables, that is, the computation of the $r(i, j)$ terms in Algorithm 1. As expected, the use of necessarily serial processing tasks such as the multiplication of the DOLs to obtain the DNF (see Example 2) limit the speedup achieved by the FPGA implementation when compared to the results obtained using the cover subtask by itself.

Although the ultimate goal of reverse engineering might be understanding complete complex biological systems, recent reported models limit their exploration over specific subsystems or punctual mechanisms, for example, the study cell-cycle regulatory network of fission yeast and apoptosis (programmed cell death) [17, 18], using a moderate number of genes between 8 and 20. Our results lead us to believe that the advantages of using the proposed architecture in reconfigurable logic will be sustained even at bigger sizes. As the sizes increase our architecture will have to rely increasingly on iterations rather than parallel computations. Nevertheless, due to the custom nature (finite field) of the data, even at these sizes the parallelism over each iteration will be enough to offer significant speedups over the alternative, fully serial implementation in general purpose processors.

Speedup is maintained at a cost of increasing resource utilization. For the considered cases, BRAMs, which are used to implement the various FIFOs and RAMs of stage 1, is the fastest growing resource. This could pose a challenge to maintain performance at even higher n values. However, this issue can potentially be resolved by complementing the BRAM capacity with a dedicated external memory bank, as is commonly included in modern reconfigurable computer platforms [19, 20]. A study of the impact of external memory on the performance of the proposed architecture is future work. An advantage of our proposed architecture is that all the required memories behave as FIFOs. Thus, their accesses

patterns are predictable and (as part of future work) we can use this information to device a buffering scheme that hides the latency of accessing the external memories.

The increase in user logic (Slices, LUTs) in higher n can be circumvented by using shallower systolic arrays (smaller d_{SA}), which still maintain competitive speedups. For example, although not shown in the table, for $n = 13$ speedups of 30× and 45× were obtained by using d_{SA} values of 64 and 128.

8.2.2. Stage 2. As explained in Section 2.2, our algorithm uses a base X_i computed in Stage 1 to determine a set of equivalence classes $D/\equiv_{X_i}$ from the original set of n-tuples. The equivalence classes are then used to compute an interpolating polynomial using (3). A greater number of variables in the chosen X_i and a high variability among the n-tuples translates to larger cardinalities of $D/\equiv_{X_i}$. The greater the cardinality of $D/\equiv_{X_i}$ the higher the number of binomials computed by (3), which constitutes the bulk of computation for Stage 2.

The timing results shown in Table 6 support the previous statements. The sets of 100 n-tuples from Stage 1 along with bases of 2 to 4 variables were input to Stage 2. Observe that since the data sets were randomly generated there was high variability in the sets of n-tuples, which resulted in similar $D/\equiv_{X_i}$ cardinalities (column *Class Reps*) and run times (columns t_F and t_C) for the various sizes of n (for a given number of variables in the chosen basis). The FPGA implementation achieves speedups of 16× to 25×, which mostly increase with n. The moderate increment in speedup as n increases is attributed to the increased benefit of parallelism for the generation of binomials, and the comparison and addition of monomials.

Resource utilization distribution in Stage 2 is roughly uniform between the user logic, that is, Slices, and BRAMs. Furthermore, the increment in resource utilization for increasing values of n and numbers of variables in the basis is quite moderate, for example, the percentage of slices goes from 8.69% for $n = 8$ to 12.87% for $n = 13$.

9. Conclusion

This paper presents a new methodology based on Lagrange interpolation with two important properties: (1) it identifies redundant variables and generates polynomials containing only nonredundant variables, and (2) it computes exclusively on a reduced data set. The analysis of the methodology for its hardware implementation led us to the identification of several reduction tasks which were generalized to a simple algorithm. The generalized algorithm can be efficiently mapped to a systolic array in which each processing cell implements a pair of binary operations between an incoming and a stored value. The tasks of Boolean cover, distinctness, and multivariate polynomial addition were implemented and served as building blocks to the rest of the application. The FPGA implementation of the reduction operations and the complete application achieved speedups of up to 172× and 67×, respectively, as compared to software implementations run on a contemporary CPU, with moderate resource utilization.

Acknowledgments

An earlier version of this paper appeared as "A systolic array based architecture for implementing multivariate polynomial interpolation tasks" in the Proceedings of the 2009 International Conference on ReConFigurable Computing and FPGAs (ReConFig'09) [21]. Dr. E. Orozco was partially supported by Grant no. P20RR016470 from the National Center for Research Resources (NCRR), a component of the National Institutes of Health (NIH). The authors thank Dr. Humberto Ortiz-Zuazaga for his helpful discussions on microarray technology and bioinformatics data trends.

References

[1] U. Muller and D. Nicolau, *Microarray Technology and Its Applications*, Springer, New York, NY, USA, 2004.

[2] P. A. Ortiz-Pineda, F. Ramírez-Gómez, J. Pérez-Ortiz et al., "Gene expression profiling of intestinal regeneration in the sea cucumber," *BMC Genomics*, vol. 10, article 262, 2009.

[3] H. De Jong, "Modeling and simulation of genetic regulatory systems: a literature review," *Journal of Computational Biology*, vol. 9, no. 1, pp. 67–103, 2002.

[4] J. F. Knabe, M. J. Schilstra, and C. L. Nehaniv, "Evolution and morphogenesis of differentiated multicellular organisms: autonomously generated diffusion gradients for positional information," in *Artificial Life XI : Proceedings of the 11th International Conference on the Simulation and Synthesis of Living Systems*, pp. 321–328, MIT Press, 2008.

[5] W. J. Blake, M. Kærn, C. R. Cantor, and J. J. Collins, "Noise in eukaryotic gene expression," *Nature*, vol. 422, no. 6932, pp. 633–637, 2003.

[6] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*, Oxford University Press, Oxford, UK, 1993.

[7] D. Bollman, O. Colón-Reyes, and E. Orozco, "Fixed points in discrete models for regulatory genetic networks," *EURASIP Journal on Bioinformatics and Systems Biology*, vol. 2007, Article ID 97356, 8 pages, 2007.

[8] R. Laubenbacher and B. Pareigis, "Equivalence Relations on Finite Dynamical Systems," *Advances in Applied Mathematics*, vol. 26, no. 3, pp. 237–251, 2001.

[9] Z. Zilic and Z. G. Vranesic, "A deterministic multivariate interpolation algorithm for small finite fields," *IEEE Transactions on Computers*, vol. 51, no. 9, pp. 1100–1105, 2002.

[10] Y. Luo, "A local multivariate Lagrange interpolation method for constructing shape functions," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 2, pp. 252–261, 2010.

[11] M.-L. T. Lee, *Analysis of Microarray Gene Expression Data*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[12] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-Based skeleton for pairwise biological sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[13] A. S. Jarrah, R. Laubenbacher, B. Stigler, and M. Stillman, "Reverse-engineering of polynomial dynamical systems,"

Advances in Applied Mathematics, vol. 39, no. 4, pp. 477–489, 2007.

[14] R. D. Leclerc, "Survival of the sparsest: robust gene networks are parsimonious," *Molecular Systems Biology*, vol. 4, article 213, 2008.

[15] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," in *Proceedings of the 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL '2000)*, pp. 91–97, May 2000.

[16] E. P. Gribomont and V. V. Dongen, "Generic systolic arrays: a methodology for systolic design," in *Proceedings of the 4th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '93)*, vol. 668 of *Lecture Notes in Computer Science*, pp. 746–761, Springer, Orsay, France, April 1993.

[17] M. I. Davidich and S. Bornholdt, "Boolean network model predicts cell cycle sequence of fission yeast," *PLoS ONE*, vol. 3, no. 2, Article ID e1672, 2008.

[18] L. Tournier and M. Chaves, "Uncovering operational interactions in genetic networks using asynchronous Boolean dynamics," *Journal of Theoretical Biology*, vol. 260, no. 2, pp. 196–209, 2009.

[19] "Convey HC-1 Datasheet," http://www.conveycomputer.com/Resources/HC-1%20Data%20Sheet.pdf.

[20] "DRC Dev System DS2000 Datasheet," http://www.drccomputer.com/pdfs/DRC_DS2000_fall07.pdf.

[21] R. A. Arce-Nazario, E. Orozco, and D. Bollman, "A systolic array based architecture for implementing multivariate polynomial interpolation tasks," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 77–82, December 2009.