Optimization of Arithmetic Datapaths with Finite Word-Length Operands *

Sivaram Gopalakrishnan[1], **Priyank Kalla**[1] and **Florian Enescu**[2]
[1]Electrical & Computer Engineering, University of Utah, Salt Lake City, UT-84112
[2]Mathematics & Statistics, Georgia State University, Atlanta, GA-30303
{sgopalak, kalla}@ece.utah.edu, fenescu@mathstat.gsu.edu

Abstract: This paper presents an approach to area optimization of arithmetic datapaths that perform polynomial computations over bit-vectors with finite widths. Examples of such designs abound in DSP for audio, video and multimedia computations where the input and output bit-vector sizes are dictated by the desired precision. A bit-vector of size m represents integer values reduced modulo $2^m(\%2^m)$. Therefore, finite word-length bit-vector arithmetic can be modeled as algebra over finite integer rings, where the bit-vector size dictates the ring cardinality. This paper demonstrates how the number-theoretic properties of finite integer rings can be exploited for optimization of bit-vector arithmetic. Along with an analytical model to estimate the implementation cost at RTL, two algorithms are presented to optimize bit-vector arithmetic. Experimental results, conducted within practical CAD settings, demonstrate significant area savings due to our approach.

I. Introduction

RTL descriptions of integer datapaths that implement polynomial arithmetic are found in many practical applications, such as in digital signal processing (DSP) for audio, video and multimedia applications [1] [2]. Such designs perform a sequence of ADD, MULT, SHIFT type of algebraic computations over bit-vectors; hence they are generally modeled at RTL or behavioural-level as *multi-variate polynomials of finite degree* [2] [3]. Initial algorithmic specifications (such as a MATLAB model) of such systems involve data representation using floating-point formats. However, they are often implemented with fixed-point architectures in order to optimize the area, delay and power related costs of the implementation [4]. Subsequently, the fixed-point model can be translated into an RTL description [5] - that can be subsequently synthesized into a circuit.

Algebraic techniques and tools have been used for synthesis and optimization of such systems. However, for their efficient and correct modeling, it is important to account for the effect of bit-vector size of the operands on the resulting computation. In other words, a bit-vector of size m represents integer values from 0 to 2^m - 1 (or integers reduced modulo 2^m). This implies that finite word-length (m) bit-vector arithmetic manifests itself as algebra over finite integer rings (Z_{2^m}). Properties of such finite rings should therefore be exploited for RTL optimization of bit-vector arithmetic.

This paper models finite word-length bit-vector arithmetic as polynomial functions (or polyfunctions) over $f : Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}} \rightarrow Z_{2^m}$. Here, $(n_1, n_2, \cdots n_d)$ are the sizes of the input bit-vectors $(x_1, x_2, \ldots, x_d)$. And, m is the size of the output bit-vector f. In other words, the computation is modeled as a multi-variate polynomial $f(x_1, \ldots, x_d)\%2^m$, where each $x_i \in Z_{2^{n_i}}$ and f is computed $\%2^m$. Properties of such polyfunctions have been analyzed in [6]. Over such finite rings, polynomials with different degrees and coefficients (*i.e.* computations with different costs) can become computationally (bit-true) equivalent. Using a cost model, along with number-theoretic properties of such rings, the paper presents two algorithms for optimization of finite word-length bit-vector arithmetic.

Motivation: Consider a computation with three inputs: $x[9:0], y[7:0]$ and $z[12:0]$, and output $F_1[15:0]$ (Eqn. 1). Another optimized implementation of the same computation is given by $F_2[15:0]$ (Eqn. 2.).

$$F_1 = 16384 * x^4 + x^3 * y + 49152 * x^2 + x * z + 1221 \quad (1)$$

$$F_2 = x^3 * y + x * z + 1221 \quad (2)$$

It should be noted that the polynomials F_1 and F_2 are symbolically distinct (polynomially, $F_1 \neq F_2$). However, due to the given bit-vector sizes, they are computationally equivalent $F_1[15:0] \equiv F_2[15:0]$. It is clear that F_2 is a cheaper implementation (area) than F_1, since it has a lower degree (fewer MULTs) and fewer monomial terms (fewer ADDs). *So, given a finite word-length bit-vector arithmetic computation $F_1[m-1:0]$, how do we derive a bit-true equivalent computation $F_2[m-1:0]$, that has a lower implementation cost?* This paper addresses the above problem by integrating finite ring algebra and number theory within a CAD-based synthesis framework.

II. Related Work

Lately, there has been increasing interest in exploring the use of algebraic manipulation for RTL synthesis of arithmetic datapaths. The works of [7] [8] derive new polynomial models of complex computational blocks for efficient synthesis. In [2], Symbolic Computer Algebra tools are used to search for a decomposition of a given polynomial according to available library elements using a Groebner's bases based approach. However, the derived polynomial models represent the computations over the fields of reals (R), fractions (Q) or over the integral domain (Z) - collectively called the *unique factorization domains* (UFDs). This often results in a polynomial approximation [3], without properly accounting for the effect of bit-vector size on the resulting computation. While the work of [9] does account for the datapath-size for allocation, it operates directly on the original (given) arithmetic expression - thus limiting the degree of freedom in searching for a better implementation.

Finite rings of the type Z_{2^m} are non-UFDs, due to the presence of nilpotent elements. (An element x of a ring is nilpotent if $x^n = 0$ for some positive integer n.) Unfortunetly, this disallows the use of fundamental computer algebra results on Euclidean division and factorization over non-UFDs. As a result, contemporary (algebra-based) high-level synthesis frameworks are limited in their capability to employ sophisticated algebraic manipulations to reduce the cost of the implementation [2].

Other algebraic transforms have also been explored for efficient hardware synthesis: factorization and common subexpression elimination [10] [11], exploiting the structure of arithmetic circuits [12], term re-writing [13], etc. However, these techniques also overlook the effect of bit-vector size on the given computation.

Note that our approach does not preclude some of the above mentioned synthesis procedures [11] [10] [9]; it can be combined

*This work has been supported in part by the following grants: 1) CAREER award, NSF CCF-546859; and 2) NSF CCF-515010; and Georgia State University (GSU) Research Initiation Grant.

1-4244-0630-7/07/$20.00 ©2007 IEEE.

with these approaches as an additional optimization step. Modulo arithmetic has been applied to the task of circuit/RTL verification [14]. The concept of polynomial functions over finite rings has also been applied to the equivalence verification of arithmetic datapaths in [15] [16]. This paper demonstrates its application to *optimization* of arithmetic datapaths.

The following two sections give the theoretical foundation required to derive algorithmic solutions to the problem addressed in this paper. The proofs of some concepts (lemmas and theorems) are provided in [6]; and hence not reproduced here.

III. Preliminary Concepts

Z corresponds to the set of integers, Z^+ to the set of non-negative integers and Z_n to the finite set of integers $\{0, 1, \ldots, n-1\}$. The ring of residue classes modulo 2^m is denoted by Z_{2^m}; where addition and multiplication are closed over $\{0, 1, \ldots, 2^m-1\}$. $Z_{2^m}[x]$ denotes the ring of univariate polynomials over the variable x, with coefficients from Z_{2^m}. Similarly, $Z_{2^m}[x_1, \ldots, x_d]$ corresponds to the ring of multivariate polynomials in d-variables, also denoted as $Z_{2^m}^d$. In the context of our work, $n_1, n_2, \ldots, n_d$ corresponds to the bit-vector sizes of the input variables $x_1, x_2, \ldots, x_d$ and m represents the output bit-vector size. Subsequently, we represent the RTL computations as polyfunctions from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}. Chen [6] defines the corresponding polyfunction as follows:

Definition III.1: A function f from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}}$ to Z_{2^m} is said to be a polynomial function (or *polyfunction*) if it is represented by a polynomial $F \in Z[x_1, x_2, \ldots, x_d]$; i.e. $f(x_1, x_2, \ldots, x_d) \equiv F(x_1, x_2, \ldots, x_d)$ for all $x_i = 0, 1, \ldots, 2^{n_i} - 1$; $i = 1, 2, \ldots, d$; (Refer to column 1 in table I).

It is possible for a polynomial with non-zero coefficients to *vanish* on such mappings; in which case the polynomial represents a *nil polyfunction* or a *vanishing polynomial* (Refer to column 2 in table I).

Henceforth, polynomial addition and multiplication are performed $\%n$ ($n = 2^m$). Also, we use the multi-index notation: $\mathbf{k} =< k_1, k_2, \ldots, k_d >$ are the (non-negative) degrees corresponding to the d input variables $\mathbf{x} =< x_1, x_2, \ldots, x_d >$, respectively.

IV. Identifying Vanishing Polynomials

We analyze the univariate polynomials that vanish on $Z_{2^m}[x]$ (for didactic purposes) and then extend the results to vanishing polynomials from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}.

Number Theory Perspective: According to a fundamental result in number theory, for any $n \in N$, $n!$ divides the product of n consecutive numbers. For example, 4! divides any 4 consecutive numbers: $99 \times 100 \times 101 \times 102$. Consequently, it is possible to find the **least** $k \in N$ such that n divides $k!$ (denoted $n|k!$). We denote this value k as $k = SF(n)$[1]. In the ring Z_{2^m}, let $SF(2^m) = k$, such that $2^m|k!$. For example, $SF(2^3) = 4$ as 8 divides 4! but 8 does not divide 3!; hence, least $k = 4$.

This property can be utilized to interpret the concept of vanishing polynomial as a divisibility issue in Z_{2^m}. If $f(x)\%2^m \equiv 0$, then $2^m|f(x)$. In $Z_{2^3}[x]$, let $8|f(x)$. But, $8|4!$ too, as $SF(8) = 4$. Therefore, if for all x, $f(x)$ can be represented as a product of 4 consecutive numbers, then $f(x)$ vanishes in Z_{2^3}. So, how can we represent a polynomial as a product of 4 consecutive numbers? The answer is: $x(x-1)(x-2)(x-3)$. Such a product expression is referred to as a *falling factorial* and is formally defined below.

Definition IV.1: Falling factorials of degree $k \in Z$ are defined according to: $Y_0(x) = 1$, $Y_1(x) = x$, $Y_2(x) = x \cdot (x-1)$, ..., $Y_k(x) = x \cdot (x-1) \cdots (x-k+1)$.(*Column 1 in table II shows the example of a univariate vanishing polynomial.*)

[1]This is a well-studied function in number theory. It was initially studied by Kempner [17] and was recently re-visited by Smarandache [18]. In recent literature, it is often referred to as the *Smarandache function* and hence we refer to it as $SF(n)$.

The above concept of falling factorials can be similarly defined for **multi-variate expressions** over $Z_{2^m}[x_1, \ldots, x_d]$:

$$\mathbf{Y_k} = \prod_{i=1}^{d} Y_{k_i}(x_i) = Y_{k_1}(x_1) \cdot Y_{k_2}(x_2) \cdots Y_{k_d}(x_d) \quad (3)$$

Extending the above concept, if a multivariate polynomial in $Z_{2^m}[x_1, \ldots, x_d]$ can be factorized into a product of $SF(2^m)$ consecutive numbers in **at least one of the variables** x_i, then it vanishes $\%2^m$. *Column 2* in table II illustrates this idea where both the input variables x_1, x_2, as well as the output F are in Z_{2^2}. We wish to extend the above concepts to analyze polynomials over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}}$ to Z_{2^m}. For this purpose, we define another quantity [6]:

$$\mu_i = min\{2^{n_i}, SF(2^m)\}; i = 1, 2, \ldots, d. \quad (4)$$

We consider the following results from [6]:

Lemma IV.1: Let $\mathbf{k} =< k_1, k_2, \ldots, k_d > \in (Z^+)^d$. Then, $\mathbf{Y_k} \equiv 0$ if and only if some $k_i \geq \mu_i$.

Column 3 in table II gives an example illustrating the application of this lemma.

When a polynomial cannot be factored into such Y_k expressions, can it still vanish? Consider the quadratic polynomial $4x^2 - 4x$ in $Z_8[x]$. It can be written as $4(x)(x-1)$. While $4x^2 - 4x$ cannot be factorized as $(x)(x-1)(x-2)(x-3)$, it still vanishes in Z_8. The missing factors, $(x-2)(x-3)$ in this case, are compensated for by the *multiplicative constant* 4; therefore, $4x^2 - 4x \equiv 0\%8$. We now need to identify the constraints on such multiplicative constants such that the given polynomial would vanish. We state the following result [6]:

Lemma IV.2: The expression $g_{\mathbf{k}} \cdot \mathbf{Y_k} \equiv 0$ if and only if $\frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k_i!)} | g_{\mathbf{k}}$; where,

$g_{\mathbf{k}} \in Z$;

$\mathbf{k} =< k_1, k_2, \ldots, k_d > \in Z^d$ such that $k_i < \mu_i, \forall i = 1, \ldots, d$.

In *column 4* of table II, we show an example where this lemma can be applied.

The above results can be extended to derive a unique canonical representation for a polynomial function from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. We state the following theorem [6]:

Theorem 1: Let F be a polynomial representation for the function f from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. Then, F can be uniquely represented as:

$$F = \Sigma_{\mathbf{k}} c_{\mathbf{k}} \mathbf{Y_k} \quad (5)$$

where, $\mathbf{Y_k}$ is the falling factorial defined in Eqn. 3;

$\mathbf{k} =< k_1, \ldots, k_d >$ for each $k_i = 0, 1, \ldots, \mu_i - 1$;

$c_{\mathbf{k}} \in Z$ such that $0 \leq c_{\mathbf{k}} < \frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k_i!)}$.

Proof: The proof is provided in [6]. Briefly reviewing it, any polynomial F from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m} can be decomposed in the form

$$F = Q_\mu \mathbf{Y}_\mu + \Sigma_{\mathbf{k}} a_{\mathbf{k}} b_{\mathbf{k}} \mathbf{Y_k} + \Sigma_{\mathbf{k}} c_{\mathbf{k}} \mathbf{Y_k} \quad (6)$$

where,

$Q_\mu \in Z[x_1, \ldots, x_d]$ is an arbitrary polynomial;

$\mathbf{Y_k}$ is the falling factorial defined in Eqn. 3;

$\mathbf{Y}_\mu = \mathbf{Y_k}$ for some $k_i \geq \mu_i$;

$\mathbf{k} =< k_1, \ldots, k_d >$ for each $k_i = 0, 1, \ldots, \mu_i - 1$;

$a_{\mathbf{k}} \in Z$ is an arbitrary integer,

$b_{\mathbf{k}} = \frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k_i!)}$ and

$c_{\mathbf{k}} \in Z$ is an arbitrary integer, such that $0 \leq c_{\mathbf{k}} < b_{\mathbf{k}}$.

It can be clearly seen from eqn. 6, that the first term ($Q_\mu \mathbf{Y}_\mu$) is a vanishing polynomial from Lemma IV.1, and the second term ($\Sigma_{\mathbf{k}} a_{\mathbf{k}} b_{\mathbf{k}} \mathbf{Y_k}$) is a vanishing polynomial from Lemma IV.2. The third term ($\Sigma_{\mathbf{k}} c_{\mathbf{k}} \mathbf{Y_k}$) cannot be reduced any further, since

TABLE I

EXAMPLES (PRELIMINARY CONCEPTS)

Example I	Example II
Let $f : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ be a polyfunction in two variables (x_1, x_2), defined as: $f(0,0) = 1$, $f(0,1) = 3$, $f(0,2) = 5$, $f(0,3) = 7$, $f(1,0) = 1$, $f(1,1) = 4$, $f(1,2) = 1$, $f(1,3) = 0$. Then, f is a polyfunction representable by $F = 1 + 2x_2 + x_1{x_2}^2$, since $f(x_1, x_2) \equiv F(x_1, x_2)\%2^3$ for $x_1 = 0, 1$ and $x_2 = 0, 1, 2, 3$.	Consider $f(x_1, x_2) : Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$ represented by the polynomial $F = 4x_1x_2^2 + 4x_1x_2$. While F has non-zero coefficients, $F\%8 \equiv 0, \forall x_1 \in Z_2, x_2 \in Z_4$.

TABLE II

EXAMPLES ILLUSTRATING THE LEMMAS (VANISHING POLYNOMIAL)

Univar. Vanishing Poly	Multivar. Vanishing Poly	Lemma IV.1	Lemma IV.2
Consider $F(x)$ over $Z_{2^3}[x]$ where $F(x) = x^4 + 6x^3 + 3x^2 + 6x$. Here $SF(2^3) = 4$. $F(x)$ can be factored as a product of 4 consecutive numbers: i.e. $(Y_4(x))$. Therefore $F(x)$ is a vanishing polynomial in $Z_{2^3}[x]$, or $F(x) \equiv Y_4(x)\%2^3$, hence $F(x)\%2^3 \equiv 0$.	Consider $F(x_1, x_2)$ over $Z_{2^2}[x_1, x_2]$ where $F(x_1, x_2) = x_1^4x_2 + 2x_1^3x_2 + 3x_1^2x_2 + 2x_1x_2$. Here, $SF(2^2) = 4$, and the highest degrees of x_1 and x_2 are $k_1 = 4$, and $k_2 = 1$, respectively. Note that $F\%4$ can be equivalently written as $F = Y_{<4,1>}(x_1, x_2)\%4$ $= Y_4(x_1) \cdot Y_1(x_2)\%4$ Since $F\%4$ can be represented as a product of 4 consecutive numbers in x_1, $2^2 \vert F$ and $F \equiv 0$	Consider $F(x_1, x_2)$ over $Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ where $F = x_1^2x_2 - x_1x_2$. Here, $SF(2^3) = 4$, $k_1 = 2$, $k_2 = 1$. $\mu_1(2^1) = min\{2^1, 4\} = 2 = k_1$ satisfying Lemma IV.1 and $\mu_2(2^2) = min\{2^2, 4\} = 4 > k_2$ F can now be written as $x_1^2x_2 - x_1x_2 \equiv x_1(x_1 - 1)x_2$ $\equiv Y_{<2,1>}(x_1, x_2)$ $\equiv 0$	Consider $F(x_1, x_2)$ over $Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ where $F = 4x_1x_2^2 + 4x_1x_2$. Here $2^{n_1} = 2$, $2^{n_2} = 4$ and $2^m = 8$ $\mathbf{k} = < k_1, k_2 > = < 1, 2 >$. So $\prod_{i=1}^{2} k_i! = 1! \cdot 2! = 2$, $SF(2^m = 8) = 4$; $\mu_1(2) = min\{2, 4\} = 2$, $\mu_2(4) = min\{4, 4\} = 4$. $F \equiv 4x_1x_2^2 + 4x_1x_2 \equiv 4 \cdot x_1 \cdot x_2 \cdot (x_2 - 1)$ $\equiv g_{<1,2>} \cdot Y_{<1,2>}(x_1, x_2) \equiv 0$ because $\frac{8}{gcd(8,1!\cdot 2!)} = 4$ which divides $g_{<1,2>} = 4$.

the coefficient $c_{\mathbf{k}} < b_{\mathbf{k}}$ and hence $b_{\mathbf{k}}$ cannot divide $c_{\mathbf{k}}$ (for Lemma IV.2 to hold true). Hence, eqn. 6 can simply be written as

$$F = \Sigma_{\mathbf{k}} c_{\mathbf{k}} \mathbf{Y_k} \tag{7}$$

■

The following example illustrates the above concept.

Example IV.1: Consider a polynomial $F = x_1^2 + 7x_1 + 6x_1x_2^2 + 6x_1x_2$ for $f : Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$. Here, $\mu_1(2) = min\{2, SF(8)\} = 2$; $\mu_2(2^2) = min\{2^2, SF(8)\} = 4$. F can be written as follows:

$$\begin{aligned} x_1^2 + 7x_1 + 4x_1x_2^2 + 4x_1x_2 &\equiv Y_{<2,0>}(x_1, x_2) + \\ a_{<1,2>}b_{<1,2>}Y_{<1,2>}(x_1, x_2) &+ c_{<1,2>}Y_{<1,2>}(x_1, x_2) \\ &\equiv c_{<1,2>}Y_{<1,2>}(x_1, x_2) \\ &\equiv 2x_1x_2^2 + 2x_1x_2 \end{aligned}$$

Here, $a_{<1,2>} = 1$, $b_{<1,2>} = 8/(8, 1! \cdot 2!) = 4$ and $c_{<1,2>} = 2$. F can be written in the form given by Theorem 1, and is the unique canonical form representation of the polynomial.

V. ALGORITHMS:POLYNOMIAL REDUCTION

In this section, we present two algorithms that use the concepts described in the earlier section to optimize a polynomial function from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}.

A. Algorithm I

Algorithm 1 takes the following inputs: Input polynomial F_1, number of variables d, variables $x_1, \ldots, x_d$ with corresponding input bit-widths $n_1, \ldots, n_d$ and the output bit-width m. The algorithm then operates as follows:

1. Assign F1 to poly and min_poly. Assign Cost(F1) (refer section VI for the cost model) to min_cost.
2. Compute value of $SF(2^m)$. It has a complexity of O(n/log n) [19]. This value is then used to obtain the μ_i values.
3. Find the max. degree (k_i) of each variable x_i in *poly*.
4. Divide the polynomial by the falling factorial expressions Y_μ in each of the d variables.
5. If the remainder is zero, it is a vanishing polynomial and the cost of the poly is zero. because $F = Q_\mu \mathbf{Y}_\mu$. Else, use the remainder as the new *poly*.
6. Update the degrees (k_i), min_cost and min_poly and continue dividing from $Y_{\mu-1}$ (highest degree) to Y_0 for each variable.
7. After each division, check for the following conditions:
   - If the quotient can be written as $a_{\mathbf{k}} \cdot b_{\mathbf{k}}$ (where $b_{\mathbf{k}}$ is defined according to Theorem 1), and the remainder is zero, return *0*. It is a vanishing polynomial.
   - If the quotient can be written as $a_{\mathbf{k}} \cdot b_{\mathbf{k}}$, and the remainder is non-zero, use the remainder as the new poly.
   - Check if the coefficient $c_{\mathbf{k}} > b_{\mathbf{k}}$. If so, perform the division with $b_{\mathbf{k}}$*$Y_{\mathbf{k}}$, and again use the remainder as the new poly.
   - Update the min_poly and continue with the next iteration.

min_poly gives the polynomial with the least cost implementation in this reduction procedure and min_cost gives its corresponding cost. The number of multi-variate divisions is bound by $O(\prod_d \mu_i)$, where μ_i is as defined previously and d is the total number of variables.

B. Algorithm II

Consider a polynomial $f = x^6 + 8x^3 + 8x$, with bit vector-sizes of {x, f} being {3, 4}, respectively. According to the previous algorithm, the reduction starts with the highest degree monomial (*highest degree* = 6, in this case) and proceeds further. Using the first algorithm, the polynomial reduction results in the following set of polynomials.

Initial Polynomial: $f = x^6 + 8x^3 + 8x$
1st Intermediate Polynomial: $f = 11 * x^5 + x^4 + 9 * x^3 + 8 * x^2 + 4 * x$
2nd Intermediate Polynomial: $f = x^5 + 11 * x^4 + 7 * x^3 + 14 * x^2$
Final Reduced Polynomial: $f = x^5 + x^4 + 3 * x^3 + 12 * x$.

Using the cost model, the initial polynomial is estimated to be the least cost polynomial (which requires only 7 multipliers, 2 constant multipliers and 2 adders). However, in this polynomial, the sub-expression $8x^3 + 8x$ is a vanishing polynomial in Z_{2^4}. Thus it can be seen that if we choose to reduce this sub-expression only, the initial polynomial f optimizes to x^6.

Initial Polynomial: $f = x^6 + 8x^3 + 8x$
(*Reduce only* $8x^3 + 8x$ and retain x^6 as is)
Optimized Polynomial: $f = x^6$.

Now, the optimized polynomial requires only 5 multipliers. Thus, using an approach, where only sub-expressions of the polynomial are reduced, the optimization is further enhanced.

To lend an algorithmic procedure to such an approach, instead of iterating over all possible degrees (refer Alg. I), we iterate over *all combinations* of all possible degrees. In other words, consider the previous example where $f = x^6 + 8x^3 + 8x$. The combination of all possible degrees is given by the set

Algorithm 1 OPT_POLY: Optimize a given polynomial.

OPT_POLY(F_1, d, $\mathbf{x}$, m, $\mathbf{n}$)
F_1 = Polynomial in $\mathbf{x}$; d = Number of variables;
$x[1 \ldots d]$ = List of input variables; m = Bit-width of F_1;
$n[1 \ldots d]$ = List of bit-widths of input variables, $\mathbf{x}$;
$poly = F_1$; $min\_cost = cost(poly)$; $min\_poly = poly$;
Compute SF(2^m);
/*Compute the values for μ_i*/
for $i = 1$ to d **do**
$\mu[i] = min\{SF(2^m), 2^{n[i]}\}$; $k[i]$ = Max. degree of $x[i]$ in $poly$;
end for
/*Check if Y_μ divides $poly$*/
for $i = 1$ to d **do**
/*Lemma IV.1*/
if $(k[i] \geq \mu[i])$ **then**
$quo, rem = \frac{poly}{Y_{<0,\ldots,k[i],\ldots,0>}(x_1,\ldots,x_d)}$;
if $(rem == 0)$ /* rem = remainder */ **then**
$min\_cost = 0$; /*$poly = Q_\mu \mathbf{Y}_\mu$; a vanishing polynomial*/
$min\_poly = 0$; return 0;
else
$poly = rem$;
Update min_poly, min_cost;
$break$;
end if
end if
end for
/*Iterate over all possible degrees*/
for $j = \prod_{l=1}^{d}(\mu_l)$ to 1 **do**
/*Update degrees*/
for $i = 1$ to d **do**
$k[i]$ = Update degree of $x[i]$ in current $poly$;
end for
$quo, rem = \frac{poly}{Y_{<k[0],\ldots,k[d]>}(x_1,\ldots,x_d)}$;
$b_{<k[0],\ldots,k[d]>} = \frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k[i]!)}$;
/*Lemma IV.2*/
if $(b_{<k[0],\ldots,k[d]>} | quo)$ **then**
if $(rem == 0)$ **then**
$min\_cost = 0$; $min\_poly = 0$; return 0;
else
$poly = rem$;
Update min_poly, min_cost;
end if
end if
c_k = Coefficient of $< k[0], \ldots, k[d] >$
/*Check for the range of the coefficient*/
if $(c_k > b_{<k[0],\ldots,k[d]>})$ /*if coefficient > the range*/ **then**
$quo, rem = \frac{poly}{b_{<k[0],\ldots,k[d]>} * Y_{<k[0],\ldots,k[d]>}(x_1,\ldots,x_d)}$;
$poly = rem$;
Update min_poly, min_cost;
end if
end for
return min_poly;

$\{(x^6+8x^3+8x),\ (x^6+8x^3),\ (8x^3+8x),\ (x^6+8x),\ (x^6),\ (8x^3),\ (8x)\}$. Each element of the set is considered as a sub-expression, and reduced [2]. It should be noted that Algorithm I is subsumed in Algorithm II, since one of the elements of the set is the entire polynomial itself. Since this is a more pervasive algorithm than the previous one, the complexity clearly increases. In this algorithm, the number of multi-variate divisions is bound by $O(\mu) = O(2^{\prod_d \mu_i})$, because in the worst-case it has to iterate through all the combinations of all degrees for every variable to determine the optimized polynomial. Using a classic branch and bound procedure, we can further optimize this search and determine the least cost polynomial. Due to lack of space, we do not provide a pseudocode for algorithm II.

[2]Note that, if there are k monomial terms, the combination set will have 2^k - 1 elements

VI. Modeling Area Cost at Polynomial Level

In the two algorithms, at every reduction step we get an *intermediate* polynomial equivalent to the original one. We wish to estimate the cost (implementation area) of the original polynomial, all intermediate polynomials and also the final reduced form and select the least cost expression for implementation.

Polynomial computations correspond to additions, multiplications and constant multiplication operations (where one input to the multiplier is a constant). For instance, consider $f = 5 * x^3 * y + 10 * x^2 * y^2 + 13 * x * y + 6 * y$. f can be implemented with 3 adders, 7 multipliers and 4 constant multipliers. If we can determine the cost of the implementation area of these modules separately, their total cost would reflect the cost of implementing the polynomial f. Hence, we model the cost of adders, multipliers and constant multipliers (implemented with finite input and output bit-vector sizes) at polynomial level.

Adders: We estimate the area of an adder based on the implementation of a ripple-carry adder. If the input bit-vector sizes of the adder are n_1 and n_2, and the output bit-vector size is m: if $Max(n_1 + 1, n_2 + 1) > m$, then we require atleast m Full Adder modules, else if $Max(n_1 + 1, n_2 + 1) < m$, then we will require $Max(n_1+1, n_2+1)$ Full Adder modules $Cost(Add) = n * Cost(FA)$ where $Cost(FA)$ is the cost of a full adder and n is the number of Full Adder modules.

Multipliers: The estimated cost of an $n_1 \times n_2$ to m-bit multiplier is modeled on an array multiplier implementation [20].

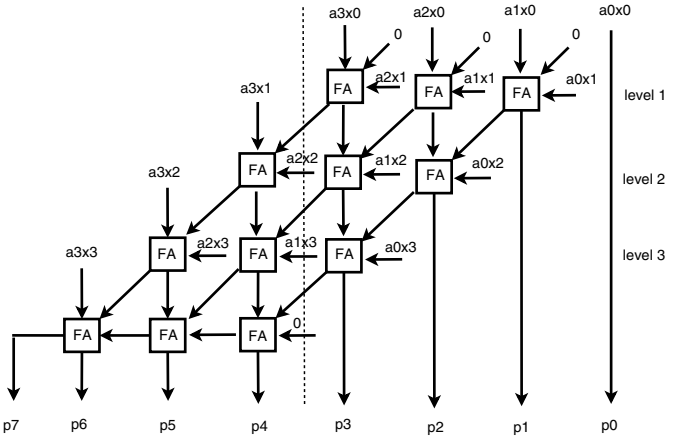

Fig. 1. Implementation of a 4-bit array multiplier (AX)

Consider the 4-bit array multiplier shown in Fig. 1. It is composed of partial product generators and an array of full adder modules. Its area can be modeled as the sum of partial product cost and the array network cost. We are interested in the area occupied by the partial products and the array network responsible for generating only the lower m output bits. For instance, in Fig. 1, if the value of m is 4, then the region of interest is to the right side of the dotted line. Therefore, the cost of the multiplier can be estimated as: $Cost(mbit\_Mult) = Cost(PP(m)) + Cost(Arr(m))$, where $Cost(PP(m))$ is the cost of partial products (implemented with AND gates) and $Cost(Arr(m))$ is the cost of the array network (implemented with FA modules). Using the structure of the array multiplier and the values of n_1, n_2 and m, we can determine the minimum number of partial products and Full adder modules required to implement an $n_1 \times n_2$ to m-bit multiplier.

Constant Multipliers: When an input to a multiplier is a constant, then the constant can be propagated to simplify the circuit. To model this effect, we need to analyze its bit pattern and estimate a cost based on the simplification caused by propagating these bits. We model constant multiplication using the array multiplier model. An $n_1 \times n_2$ to m-bit constant multiplier is modeled as an $m \times m$ to m-bit multiplier (by either padding

0's or truncation), to apply our constant propagation strategy. In other words, if n_1 (or n_2) is smaller than m, then the remaining bits (m - n_1, (or $m - n_2$)) are padded with zeroes till the m-th bit. If n_1 (or n_2) is greater than m, then only the lower order m-bits (from n_1, and n_2) are chosen for the implementation. In this manner, for an $m \times m$ to m-bit multiplier, only the lower order m-bits are analyzed for constant propagation.

Simplification using Constant Propagation: In figure 1, consider X as the constant and A as the variable. To propagate the constant X, we analyze the bits from the least significant position ($X[0]$) to the most significant one ($X[m-1]$). Here are some results that we have derived to estimate the area as a result of constant propagation.

1. *While traversing X from its LSB to MSB, until we reach a bit position whose value is* 1*, the cost of the implementation is zero due to zero propagation:*

Consider the bit-pattern of $X = \{X[m-1], X[m-2], ..., X[i] = 1, 0, 0..., X[0] = 0\}$. Here, $X[i]$ is the least significant bit with value 1. The partial products generated using $X[k]$, $k < i$ will be 0. Therefore, up to the i-th level, 0s are fed into the full adder modules, which results in their complete elimination (simplification) upto (i-1) levels.

2. *Until we reach the second bit position with value* 1 *in X while traversing from its LSB to MSB, the cost of the implementation is still zero:*

Consider the bit-pattern of $X = \{X[m-1], ..., X[k] = 1, 0, ..., X[i] = 1, 0, ..., X[0] = 0\}$. Here $X[i]$ is least bit position with value 1 and $X[k]$ is the next least bit position with value 1. We know that area cost due to the bits from $X[0]$ to $X[i-1]$ is zero from the previous result. The partial products generated by $X[i]$ keep propagating until the kth level because: a) there are no carry signals generated in the ith level; and b) every subsequent level until (k-1) performs an addition with 0 (partial products due to $X[i+1]$ to $X[k-1]$ are 0).

3. *On encountering the second bit position with value* 1 *in the traversal of X from its LSB to MSB, the full adder modules in that level can be optimized to half adder modules:*

Consider the bit pattern used in the previous result. The partial products generated by $X[i]$ and $X[k]$ are added at the kth level. However, the carry-signals feeding the full adder modules in the k-th level are 0. Hence these can be optimized to half adder modules.

4. *For the subsequent levels, if the value of $X[i]$ at any level is* 0*, then the full adders in that level reduce to half adders:*

Since the partial products generated due to $X[i]'s = 0$ are also 0, the full adders being fed by these partial products are simplified to half adders.

Based on the bit pattern of the constants, the above models are employed to estimate the effect of constant propagation on the multiplier area.

Example: Consider the effect of $3 * A$ and $5 * A$ in a multiplier with output bit-vector size m=4. Figures 2 and 3 depict the optimization in the designs for the multiply operation with constants 3 and 5, respectively.

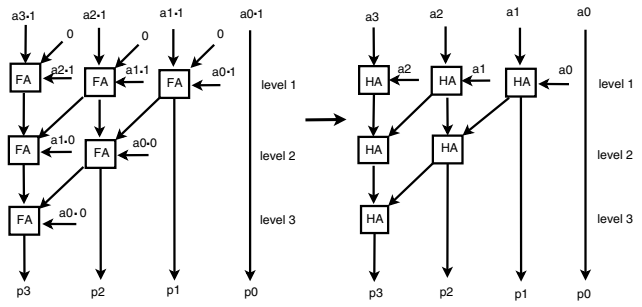

Fig. 2. Implementation of 3A, X=0011

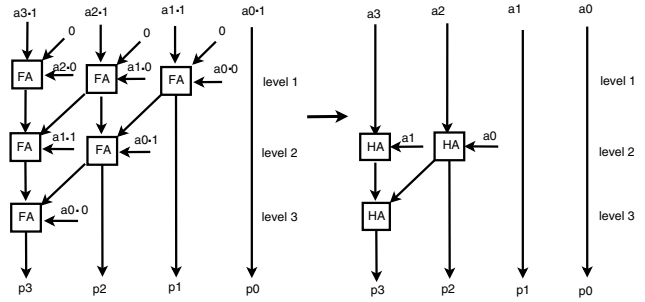

Fig. 3. Implementation of 5A, X=0101

Quantifying the cost We employ the unit model cost, where every logic gate can be implemented with a unit cost ,to *quantitatively* calculate the area of the polynomial.

VII. Experiments

The algorithms were implemented in Perl with calls to Maple [21], along with the presented cost model, for optimizing the given polynomial. The polynomial representing the datapath and the operating bit-vector size (*input/output* - $n_1, n_2, ...n_d/m$) were given as the inputs to the tool. Step-by-step reductions of the given polynomial were performed using our algorithms until a minimal form was obtained. For the original, minimal and every intermediate polynomial generated, the implementation cost was estimated. The polynomial with the least estimated cost was selected for implementation.

We used the Synopsys Design Compiler to generate the required $n_1 \times n_2$ to m-bit adders and multipliers. These units were used, subsequently, as functional units to implement the polynomials. To compare the area statistics, both the original polynomial and the reduced polynomial with least estimated cost were implemented using the Synopsys Module Compiler.

Experiments have been performed on a variety of DSP benchmarks and the results are presented in Table III. The first four examples are from [12]. Deg4, Janez and Cubic are polynomial filters used in image processing applications [1]. IRR is an image rejection receiver from [22]. Mibench is an automotive application from [23]. Antialias and PSK (phase shift keying) are from [2], and IIR-4 is a 4th order IIR computation. Column 2 lists the design characteristics: number of variables, their highest degree and the bit-vector sizes of the inputs/output ($n_1, ..n_d/m$). Column 3 lists the estimated cost of the original polynomial. Column 4 and 5 list the cost of the optimized polynomial using algorithm I (Alg1) and algorithm II (Alg2), respectively. In columns 6 and 7, we list percentage improvement obtained in the estimated cost using Alg1 (Imp1) and Alg2 (Imp2), respectively. For the implementation cost, we report the results of Alg2. Column 8 and 10 list the actual implementation area of the original and the selected polynomial (synthesized), respectively. Column 9 and 11 depict the *critical path delay* of the original and the selected polynomial implementations respectively. These implementations have been realized using shifters, multipliers and adders. Column 12 depicts the improvement in the area of the implementation while column 13 depicts the improvement in the critical path delay in the implementation. If the improvement in the estimated cost is less than 1%, we choose the original polynomial for implementation.

For the first 9 benchmarks, we are able to find a reduced implementation. There is an average improvement of approximately 34% in actual implementation. For the remaining benchmarks, our cost estimate provided an improvement of less than 1% and hence, the original polynomial was chosen for implementation. Considering all the benchmarks, the average improvement in the actual implementation area is still approximately 23%.

Expression manipulations: There are many expression ma-

TABLE III

COMPARISON OF PERFORMANCE OF THE ESTIMATION AND IMPLEMENTATION COSTS

Benchmark	Var/Deg/	Estimated Cost					Implementation Cost					
	$n_1,..n_d/m$	Orig	Alg1	Alg2	Imp.1 %	Imp.2 %	Original		Alg2		Improv. %	
							Area	Delay	Area	Delay	Area	Delay
Poly1	3/4/14, 14, 16/16	7581	3927	3766	**48.2**	**50.3**	37430	372.45	20628	288.63	**44**	22.5
Poly2	3/4/10, 8, 13/16	4820	2393	2393	**50.3**	**50.3**	28848	288.63	11684	214.35	**59.49**	25.73
Poly3	2/5/13, 13/16	6227	5465	5465	**11.7**	**11.7**	28840	335.32	23006	298.18	**20.2**	11.07
Poly_unopt	1/4/12/16	5196	2994	2994	**42.3**	**42.3**	28836	335.32	14424	214.35	**49.9**	36.07
Deg4	3/4/16, 8, 16/16	22731	16361	16361	**28**	**28**	116684	632.44	82718	521.02	**29.1**	17.61
Janez	1/5/12/16	8907	6163	6154	**30.8**	**30.9**	42910	372.45	28840	335.32	**32.7**	9.97
Mibench	2/9/16,12/16	58510	48226	48226	**17.6**	**17.6**	249290	977.31	216772	921.07	**13.04**	5.75
IRR	2/4/16,8/16	10864	6943	6811	**36.1**	**37.3**	54594	400.04	37792	362.91	**30.77**	9.28
Antialias	1/7/11/16	15997	12011	12011	**24.9**	**24.9**	79254	540.12	59712	502.98	**24.65**	6.87
PSK	2/4/11,14/16	18140	18140	18140	**<1%**	**<1%**	76876	-	-	-	-	-
Cubic	3/3/24,28,31/32	47595	47586	47586	**<1%**	**<1%**	256388	-	-	-	-	-
IIR-4	2/4/24,29/32	49339	49333	49333	**<1%**	**<1%**	213408	-	-	-	-	-

nipulation techniques that have been used to optimize arithmetic datapaths such as factorization, tree-height reduction, horner implementation and common-subexpression elimination. While such techniques are commonly used in synthesis of arithmetic polynomials, our approach can be used as a *pre-processing step*, thus providing an additional scope for optimization. High-level synthesis techniques such as scheduling and resource-sharing can also be employed to reduce the number of components and improve the critical path in an arithmetic expression.

All the techniques mentioned above operate on the given data-flow graph (computation) and will still need to implement all the operations shown in that graph. On the other hand, the data-flow graph generated by our approach leads to a better implementation. This graph can be further optimized by expression manipulation, scheduling and resource-sharing.

Limitation of our approach: Given a polynomial f of degree **k**, one can derive a vanishing polynomial q of higher degrees (say, **k+1**) too. By computing $f + q$, one can create a higher degree (**k+1**) polynomial equivalent to f. The cost of $f + q$ might be cheaper than f. Our approach cannot identify cheaper implementations of a higher degree. Unfortunately, there can be more than one vanishing expression of a given degree (depending upon the coefficients) that can be added to f. This makes it difficult to derive a "convergent" algorithm to search for low-cost implementations of higher degree.

VIII. Conclusions and Future Work

This paper has presented an area optimization approach for polynomial datapaths: where the input and output bit-vector sizes of the operands are given as $(n_1, n_2, ..., n_d)$ and (m), respectively. Finite word-length bit vector arithmetic is then modeled as a polyfunction from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}. Exploiting the concept of vanishing polynomials over this mapping, we present two algorithms to optimize a given polynomial to a polynomial with low cost implementation. A cost model to estimate the area at polynomial level is also presented. Using the optimization procedure, along with the cost model, allows to select an equivalent lower cost expression for synthesis. Experiments show significant area savings using our approach. Also, it can be seen that the area savings do not worsen timing. We are currently investigating how to extend our approach to perform polynomial decompositions over such arithmetic.

References

[1] V. J. Mathews and G. L. Sicuranza, *Polynomial Signal Processing*, Wiley-Interscience, 2000.

[2] A. Peymandoust and G. DeMicheli, "Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis", *IEEE Trans. CAD*, vol. 22, pp. 1154–11656, 2003.

[3] J. Smith and G. DeMicheli, "Polynomial circuit models for component matching in high-level synthesis", *IEEE Trans. VLSI*, vol. 9, 2001.

[4] D. Menard, D Chillet, F Charot, and O. Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation", *in Intl. Conf. Compiler, Architecture, Synthesis Embedded Sys., CASES*, 2002.

[5] I. A. Groute and K. Keane, "M(VH)DL: A MATLAB to VHDL Conversion Toolbox for Digital Control", *in IFAC Symp. on Computer-Aided Control System Design*, Sept. 2000.

[6] Z. Chen, "On polynomial functions from $Z_{n_1} \times Z_{n_2} \times \cdots \times Z_{n_r}$ to Z_m", *Discrete Math.*, vol. 162, pp. 67–76, 1996.

[7] J. Smith and G. DeMicheli, "Polynomial methods for component maching and verification", *in In Proc. ICCAD*, 1998.

[8] J. Smith and G. DeMicheli, "Polynomial methods for allocating complex components", *in Proc. DATE*, 1999.

[9] G. Constantinides, P. Cheung, and W. Luk, "Heuristic Datapath Allocation for Multiple Wordlength Systems", 2001.

[10] A. Hosangadi, F. Fallah, and R. Kastner, "Factoring and eliminating common subexpressions in polynomial expressions", *in ICCAD*, pp. 169–174, 2004.

[11] A. Hosangadi, F. Fallah, and R. Kastner, "Energy Efficient Hardware Synthesis of Polynomial Expressions", *in Int'l. Conf. on VLSI Design*, pp. pp. 653–658, 2005.

[12] A. K. Verma and P. Ienne, "Improved use of the Carry-save Representation for the Synthesis of Complex Arithmetic Circuits", *in Proceedings of the International Conference on Computer Aided Design*, 2004.

[13] Arvind and X. Shen, "Using term rewriting systems to design and verify processors", *IEEE Micro*, vol. 19, pp. 36–46, 1998.

[14] C.-Y. Huang and K.-T. Cheng, "Using Word-Level ATPG and Modular Arithmetic Constraint Solving Techniques for Assertion Property Checking", *IEEE Trans. CAD*, vol. 20, pp. 381–391, 2001.

[15] N. Shekhar, P. Kalla, F. Enescu, and S. Gopalakrishnan, "Equivalence Verification of Polynomial Datapaths with Fixed-Size Bit-Vectors using Finite Ring Algebra", *in Intl. Conf. on Computer-Aided Design, ICCAD*, 2005.

[16] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Arithmetic Datapaths with Multiple Word-Length Operands", *in Proc. DATE*, 2006.

[17] A. J. Kempner, "Polynomials and their residual systems", *Amer. Math. Soc. Trans.*, vol. 22, pp. 240–288, 1921.

[18] F. Smarandache, "A function in number theory", *Analele Univ. Timisoara, Fascicle 1*, vol. XVII, pp. 79–88, 1980.

[19] D. Power, S. Tabirca, and T. Tabirca, "Java Concurrent Program for the Smarandache Function", *Smarandache Notions Journal*, vol. 13, pp. 72–84, 2002.

[20] I. Koren, *Computer Arithmetic Algorithms*, A. K. Peters, 2002.

[21] Maple, ", http://www.maplesoft.com.

[22] C. Chen and C. Huang, "On the Architecture and Performance of a Hybrid Image Rejection Receiver", *IEEE Journal on Selected Areas in Communication*, vol. 19, pp. 1029–1040, Jun, 2001.

[23] M. R. Guthaus and *et al.*, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite", *in IEEE 4th Annual Workshop on Workload Characterization*, Dec 2001.