Polynomial Datapath Optimization using Constraint Solving and Formal Modelling

Finn Haedicke†, Bijan Alizadeh‡, Görschwin Fey†, Masahiro Fujita‡, Rolf Drechsler†
† Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{finn, fey, drechsle}@informatik.uni.bremen.de
‡ VLSI Design and Education Center (VDEC), University of Tokyo and CREST, Tokyo, Japan
alizadeh@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

***Abstract*—For a variety of signal processing applications polynomials are implemented in circuits. Recent work on polynomial datapath optimization achieved significant reductions of hardware cost as well as delay compared to previous approaches like Horner form or *Common Sub-expression Elimination* (CSE). This work 1) proposes a formal model for single- and multi-polynomial factorization and 2) handles optimization as a constraint solving problem using an explicit cost function. By this, optimal datapath implementations with respect to the cost function are determined. Compared to recent state-of-the-art heuristics an average reduction of area and critical path delay is achieved.**

I. Introduction

Although polynomial expressions are frequently encountered in many applications such as computer graphics and *Digital Signal Processing* (DSP) domains, conventional high-level synthesis techniques are not able to manipulate polynomial expressions efficiently due to the lack of suitable optimization techniques for redundancy elimination over $\mathbb{Z}_{2^n}$. From a synthesis point of view, the designers often optimize such polynomial functions manually to achieve efficient *Register-Transfer-Level* (RTL) implementation. However this process can be both time consuming and error prone. As a result, developing high-level optimization and synthesis techniques is desirable to automate the design of custom datapaths from a behavioral description.

The design of computationally expensive embedded systems for multimedia and DSP applications starts with the algorithmic specification in a high level language such as MATLAB. This specification performs a sequence of arithmetic polynomial computations with integer variables of infinite bit-widths which is often implemented with fixed-point architectures. When refining algorithmic specifications to RTL descriptions, the RTL models are often implemented with fixed word-length datapath architectures. The polynomial computations are carried out over n-bit integers where the size of the entire datapath is kept constant by signal truncation. Hence modular polynomial optimization and synthesis should be supported. For example, consider $f_1 = x+5y$, $f_2 = 5x^2-9y^2$ and $f_3 = x^4+6x^3+12x^2+6x-5y^2$ which are not equivalent and also do not have any common sub-expression over $\mathbb{Z}$. After computing $f_1 = (x+5y) \bmod 4 = x+y$, $f_2 = (5x^2-9y^2) \bmod 4 = x^2-y^2 = (x+y)(x-y)$ and $f_3 = x^4+6x^3+12x^2+6x-5y^2 \bmod 4 = x^2-y^2 = (x+y)(x-y)$ using Modular-*Horner Expansion Diagrams* (HED) [1], it is obvious that ($f_2 \bmod 4 = f_3 \bmod 4$) and a common term, i.e. $x+y$, exists over $\mathbb{Z}_4$. This common term can be shared between the implementations of f_1, f_2 and f_3.

A. Our Contributions

Let $f_1(\vec{x}), \ldots, f_s(\vec{x})$ be s given polynomial functions over $\mathbb{Z}_{2^n}$ where $\vec{x} = (x_1, x_2, \ldots, x_d)$ is a vector of d input variables and n is the word-length of each variable. This paper concentrates on finding factorizations of these functions so that a maximal number of monomials is shared between the s given polynomial functions over $\mathbb{Z}_{2^n}$ in order to optimize the area as much as possible. We propose an algorithm that is able to symbolically factorize a set of functions, $f_i(i = 1, \ldots, s)$, by factorizing f_i into three sub-polynomials $p_{1,i}$, $p_{2,i}$ and $p_{3,i}$ so that $f_i = p_{1,i} \times p_{2,i} + p_{3,i}$, where coefficients are later matched to new, less expensive values. This optimization is done with respect to a cost function, that estimates area in terms of multipliers or adders. While the structure of the decomposition is similar to previous heuristics, in conclusion, our main contributions in this paper are as follows:

- Using symbolic factorization
- Minimization using formal methods
- Optimal results with respect to the cost function

B. Structure

This paper is structured as follows. The next two sections give an overview of related work in the area of polynomial minimization and introduce some definitions and notations used in this paper, respectively. Afterwards in Section IV and Section V the single- and multi-polynomial optimization algorithms are described. Experimental results are presented in Section VI followed by a summary of this work in Section VII.

II. Related Work

Although a lot of work has been done to perform optimizations in the context of code generation techniques [2], the presented algorithms do not efficiently reduce the number of operations in a set of arithmetic expressions. Some work was done to optimize code with arithmetic expressions by factorization of the expressions [3]. This work developed a canonical form for representing the arithmetic expressions and an algorithm for finding common sub-expressions. The drawback of this algorithm is that it takes advantage of only the associative and commutative properties of arithmetic operators and therefore can only optimize expressions consisting of a single type of associative and/or commutative operator at a time. As a result, it cannot generate complex common sub-expressions consisting of additions and multiplications. Moreover, this approach does not provide a modular factorization over $\mathbb{Z}_{2^n}$.

The Horner form is a well-known representation of polynomial expressions and is the most straightforward way of evaluating polynomial approximations of trigonometric functions in many libraries. This method transforms the expression into a sequence of nested additions and multiplications, which are suitable for sequential machine evaluation using *Multiply-Accumulator* (MAC) instructions. In spite of its advantages in sequential implementations, it does not provide an efficient optimization for combinational multivariate polynomials.

This reseach work was supported in part by the German Academic Exchange Service (DAAD) under grant number D/09/02091.

As an example, using the Horner form, the polynomial $2x^2z+6xyz$ is factorized as $x(2xz+6yz)$.

Symbolic computer algebra based manipulation [3], [4], [5] and factorization with *Common Sub-expression Elimination* (CSE) [6] are much better approaches in optimizing polynomials compared to the Horner form. For instance, the function $2x^2z+6xyz$ is reduced to $xz(2x+6y)$, using the CSE method. We can enhance this approach with a coefficient factorization to obtain $2xz(x+3y)$. Moreover, CSE can be combined with the Modular-HED to provide more efficient polynomials over $\mathbb{Z}_{2^n}$ [1]. Despite these advantages, the CSE method is unable to efficiently factorize some sort of polynomials. For example $x^2+6xy+9y^2$ is factorized as $x(x+6y)+9y^2$ if we employ the kernel/co-kernel extraction with CSE [6], [7]. A better factorization for this polynomial is $(x+3y)^2$.

The approximate factorization algorithm in [6] is an efficient approach in representing an arithmetic function f as a product of sub-functions: $f=f_1\times f_2\times\ldots\times f_N$, where f_i is a multivariate polynomial. However, this approach is able to factorize square-free polynomials and cannot deal with a sub-function f_i with a degree higher than one. An example is the polynomial $(x+3y)^2$, which includes a sub-function with the degree of two. Regarding such polynomials, the method in [8] has to be enhanced with techniques to initially reduce the degree of all the sub-functions to one. Another important drawback of this method is that it cannot handle those polynomials which are not reducible to $f=f_1\times f_2\times\ldots\times f_N$. As an example the function $x^2+6xy+9y^2+2z$ cannot be reduced to the function $(x+3y)^2+2z$ using [8], because we need to leave out the monomial z. There have been some efforts in the area of polynomial optimization. However, they are limited in their capability to employ sophisticated manipulations to reduce the cost of the implementation.

Algebraic techniques in [9], [10] employed various optimization techniques to manipulate the polynomials and extract common sub-expressions. The technique in [10] first extracts coefficient multiplications. Then using the kernel/co-kernel extraction techniques from [7] and [11], common cubes are extracted. Using these extraction techniques, a large number of linear blocks is exposed. Finally, algebraic division is performed to determine whether the obtained linear blocks are good divisors for optimizing the hardware implementation. Despite these advantages, this technique is only applicable to those polynomials in which linear blocks exist explicitly. For example, this technique is not able to decompose x^3+y^3 because $(x+y)$ does not exist in the given polynomial as a linear term.

Another algebraic method has been proposed in [12] and then improved in [13]. The main idea is similar to algebraic division techniques used in logic synthesis. This technique tries to decompose the original polynomial $poly$ as $p_1\times p_2+p_3$ while p_3 should be minimized. For doing so, all possible initial values of p_1 and p_2 must be evaluated. Then for each initialization it is necessary to check whether other monomials in $poly$ can be represented in the form $p_1\times p_2$. Finally, the best initialization, which constitutes the lowest complexity p_3, is chosen.

The presented paper abstracts the optimization heuristics in [12], [13] to employ a symbolic partitioning algorithm and to use formal methods to find a minimal factorization with respect to the cost function. The algorithm is later extended to work on multiple polynomials.

III. Preliminaries

This section defines the terms necessary for subsequent parts of the paper. The following definitions of monomial, term and polynomial will be used.

Definition 1: $m=\prod_{j=1}^{d}x_j^{P_j}$ is called a *monomial* with d variables, where x_j is an input variable, P_j is the degree of x_j in m, and P_j is a non-negative integer value.

Definition 2: $poly=\sum_{i=1}^{M}k_i\times m_i=\sum_{i=1}^{M}t_i$ is called a multivariate *polynomial* with M *terms* ($t_i=k_i\times m_i$), where k_i and m_i are a constant coefficient and a monomial, respectively.

To work on elements of polynomials in the above representation or in other representations, e.g. factorized, some polynomial-specific sets are defined.

Definition 3 (T_{poly}, T_{poly}^L, M_{poly}, M_{poly}^L): Given a polynomial in the form $poly=\sum_{i=1}^{M}t_i$ the set $T_{poly}=\{t_1,\ldots,t_M\}$ contains all terms in the polynomial. For a polynomial $poly'$ in another representation the literal terms $T_{poly'}^L$ containing terms t_i and sub-polynomials p_j,p_k are defined inductively based on the syntactic structure:

$$\begin{aligned}T_{t_i}^L &:= \{t_i\}\\ T_{p_j+p_k}^L &:= T_{p_j}^L\cup T_{p_k}^L\\ T_{p_j*p_k}^L &:= T_{p_j}^L\cup T_{p_k}^L\end{aligned}$$

The sets $M_{poly}:=\{m|(k,m)\in T_{poly}\}$ and $M_{poly}^L:=\{m|(k,m)\in T_{poly}^L\}$ contain all monomials and literal monomials in $poly$, respectively.

In this work it is necessary to know the coefficient of a monomial in a polynomial.

Definition 4 ($\mathrm{coeff}(poly,m)$): Given a polynomial $poly=\sum_{i=1}^{M}k_i\times m_i$,

$$\mathrm{coeff}(poly,m):=\begin{cases}k & (k,m)\in T_{poly}\\ 0 & \text{otherwise}\end{cases}$$

denotes the *coefficient of* m in $poly$.
This means $\mathrm{coeff}(x^3+3x,x)=3$ and $\mathrm{coeff}(-4x^4+(a+b)x^2y,x^2y)=a+b$.

Definition 5 ($C(m)$, $C(poly)$): The *complexity* of a given monomial $m_i=\prod_{j=1}^{d}x_j^{P_{i,j}}$ is denoted by $C(m_i)=\sum_{j=1}^{d}P_{i,j}$. The *complexity* of a polynomial $poly$ is the highest complexity of a monomial in $poly$: $C(poly)=\max_{m\in M_{poly}}C(m)$.
In other words, the complexity of a monomial is the number of variable usages: $C(x^2y^3)=C(xxyyy)=5$.

Definition 6 ($\mathrm{sub}(m)$): For any monomial $m=\prod_{j=1}^{d}x_j^{P_j}$ the set

$$\mathrm{sub}(m)=\left\{m'\middle|m'=\prod_{j=1}^{d}x_j^{P'_j}\wedge\forall j\in\{1,\ldots,d\}:P'_j\leq P_j\right\}$$

contains all *sub-monomials* of m_i (including m_i).
The set of all sub-monomials for x^2y^2z is

$$\begin{aligned}\mathrm{sub}(x^2y^2z) = \{&1,x,x^2,x^2y,x^2y^2,x^2y^2z,x^2yz,x^2z,xy,xy^2,\\ &xy^2z,xyz,xz,y,y^2z,yz,z\}\end{aligned}$$

Therefore $\mathrm{sub}(m)$ contains $\prod_{j=0}^{d}(P_j+1)$ elements for any monomial $m=\prod_{j=1}^{d}x_j^{P_j}$.

Definition 7: Given two sets A,B of monomials the set $A\times B=\{m_a\times m_b|m_a\in A\wedge m_b\in B\}$ is called *monomial set multiplication*.

Furthermore the notation $z=\mathrm{ite}(b,x,y)$ is used as abbreviation for the if-then-else expression:

$$z=\mathrm{ite}(b,x,y)=\begin{cases}x & \text{if } b\\ y & \text{otherwise}\end{cases}$$

IV. Single-Output Polynomial Optimization Algorithm

The algorithm works on a symbolic representation of polynomials in terms of constraints. By this a constraint solver can be applied to find equivalent polynomials. A cost function guides the solver in finding area efficient solutions. Just like the polynomial, the

cost function is represented by constraints, counting the number of multipliers or adders respectively.

This section explains the steps for the formal polynomial optimization algorithm. The basic idea is to recursively decompose a given polynomial, *poly*, into three sub-polynomials, p_1, p_2 and p_3, such that $poly = p_1 \times p_2 + p_3$. As a decomposition into $p_1 \times p_2$ is not always possible, the compensation term, p_3, is subtracted from the original polynomial to make it decomposable as $poly - p_3 = p_1 \times p_2$. For this purpose, we propose an algorithm that consists of multiple steps. These steps are:

(1) Create a symbolic factorization $poly = p_1 \times p_2 + p_3$ that represents all possible factorizations.
(2) Derive formal constraints describing the symbolic factorization.
(3) Add constraints describing the cost function.
(4) Use a constraint solver to find a minimal factorization with respect to the cost function.

The following subsections describe the factorization steps.

A. Symbolic Factorization

As a first step a symbolically factorized polynomial

$$poly_{sym} = p_1 \times p_2 + p_3 \tag{1}$$

is created. The polynomial $poly_{sym}$ is symbolic as all coefficients in p_1, p_2, p_3 are free variables. A valuation of the coefficients of $poly_{sym}$ yields an equivalent, factorized form of *poly*.

To create all monomials in *poly* a naïve way is to take the set of all monomials in *poly* M,

$$M = \bigcup_{m \in M_{poly}} \text{sub}(m) \tag{2}$$

as a basis for p_i. But as p_1 and p_2 in Equation (1) are multiplied, the complexity of the resulting monomials would exceed those in *poly*. Therefore a filter is applied to remove the most expensive monomials:

$$\text{filter}(M) = \left\{ m \middle| m \in M \wedge C(m) < \max_{m' \in M} C(m') \right\}$$

This filter removes all monomials with the highest complexity, which may be only a single one in the simplest case. If multiple monomials have the highest complexity all of them are removed. Removing only the top monomials guarantees that all factorizations, e.g. Horner form, are still represented by $poly_{sym}$. Later even stricter filters are used to improve the run-time. With this filter function the sets of monomials $M_{1,2} = \text{filter}(M)$ and $M_3 = M_{1,2} \times M_{1,2}$ are built. The symbolic representation is then created using the free variables a_i, b_j and c_k for p_1, p_2 and p_3, respectively:

$$\begin{aligned} poly_{sym} &= p_1 \times p_2 + p_3 \\ &= (\sum_{m \in M_{1,2}} a_i m) \times (\sum_{n \in M_{1,2}} b_j n) + \sum_{o \in M_3} c_k o \qquad (3) \\ &= (a_1 b_1 m_1 + a_1 b_2 m_2 + \ldots + a_2 b_1 m_2 + \ldots) \\ &\quad + c_1 m_1 + c_2 m_2 + \ldots \\ &= (a_1 b_1 + c_1) m_1 + (a_1 b_2 + a_2 b_1 + c_2) m_2 \\ &\quad + \ldots \qquad (4) \end{aligned}$$

B. Factorization Constraints

Given Equation (4), the constraints for a valid factorization can be derived. For each monomial m_i the coefficients in the non-factorized form of $poly_{sym}$ must match the coefficient of m_i in *poly*. For this the symbolic representation in $poly_{sym}$ is tied to the value in *poly*:

$$\forall (\phi \times m) \in T_{poly_{sym}} : \phi = \text{coeff}(poly, m) \tag{5}$$

Algorithm 1: Symbolic partitioning algorithm (factorize_terminal)

Input: Set of Monomials M
Output: Map from monomial to symbolic sum representing the factorization

1 $M_{1,2} \leftarrow \text{filter}(M)$;
2 $M_3 \leftarrow M_{1,2} \times M_{1,2}$;
3 $symbolic \leftarrow \emptyset$;
4 **foreach** $m \in M_{1,2}$ **do**
5 **foreach** $n \in M_{1,2}$ **do**
6 | $m_i \leftarrow m \times n$;
7 | **if** $(m_i, \phi) \in symbolic$ **then** update $symbolic[m_i] = \phi + a_i b_j$;
8 | **else** set $symbolic[m_i] = a_i b_j$;
9 **foreach** $o \in M_3$ **do**
10 | **if** $(o, \phi) \in symbolic$ **then** update $symbolic[o] = \phi + c_k$;
11 | **else** set $symbolic[o] = c_k$;
12 **return** *symbolic*;

Which can be expanded to match Equation (4):

$$\begin{aligned} a_1 b_1 + c_1 &= \text{coeff}(poly, m_1) \\ \wedge \quad a_1 b_2 + a_2 b_1 + c_2 &= \text{coeff}(poly, m_2) \\ \wedge \quad \ldots &= \ldots \end{aligned}$$

This means any valid assignment of a_i, b_j and c_k following the constraint in Equation (5) yields a valid factorization of *poly* in Equation (3).

C. Algorithm

Algorithm 1 creates the symbolic representation. The input to the algorithm is the set M of all sub-monomials in *poly* as defined in Equation (2) and the return value is the map named *symbolic* from monomials in $poly_{sym}$ to their respective symbolic coefficients expression as in Equation (4).

In lines 1-2 the input will first be filtered and expensive monomials are removed. This is done as p_1 and p_2 are to be multiplied and the complexity as well as the number of terms of the resulting polynomial is reduced without losing any expressiveness. The remaining monomials are added to $M_{1,2}$. Afterwards M_3 is calculated as $M_3 = M_{1,2} \times M_{1,2}$.

The double loop in line 4 and line 5 incrementally creates the product $p_1 \times p_2$. In each loop one monomial is calculated (line 6) and the sum of coefficients for this monomial is updated (line 7 or line 8, if a partial symbolic coefficient ϕ was already calculated or this one is new). For p_3 the loop in line 9 does the same with the monomials in M_3.

D. Recursive Factorization

The partitioning algorithm can also be applied recursively to find less expensive factorizations, resulting e.g. in

$$\begin{aligned} poly &= (p_{1,1} \times p_{1,2} + p_{1,3}) \times (p_{2,1} \times p_{2,2} + p_{2,3}) \\ &\quad + (p_{3,1} \times p_{3,2} + p_{3,3}) \end{aligned} \tag{6}$$

where $p_{i,j}$ are either recursive themselves or terminal. Similar to the simple symbolic factorization algorithm the complexity of $p_{1,j}$ and $p_{2,j}$ is reduced in each step. The filter reduces the complexity of p_1 and p_2 by 1 in so their recursion ends in a linear number of steps. For p_3 additionally a recursion limit d is used so that the algorithm is guaranteed to terminate in depth d. Using this approach, a symbolic factorization tree is built, as depicted in Figure 1. The nodes in the last level are called *terminal* and calculated according to the previous section. The nodes on higher levels are called *recursive* and compose three nodes from a lower level into a symbolic factorization according to Equation (6).

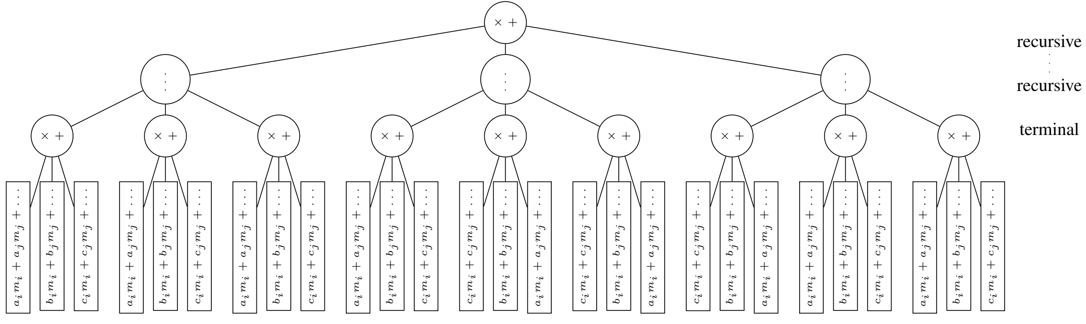

Figure 1. Recursive partitioning scheme

Algorithm 2: Symbolic partitioning algorithm (factorize_recursive)

Input: Set of Monomials M, max recursion depth d
Output: Map from monomial to symbolic sum representing the factorization

1 $M_{1,2} \leftarrow \text{filter}(M)$;
2 $M_3 \leftarrow M_{1,2} \times M_{1,2}$;
3 **if** $\max_{m' \in M} C(m') > 1 \wedge d > 1$ **then**
4 │ $symb_1$ = factorize_recursive($M_{1,2}$, $d-1$);
5 │ $symb_2$ = factorize_recursive($M_{1,2}$, $d-1$);
6 │ $symb_3$ = factorize_recursive(M_3, $d-1$);
7 **else**
8 │ $symb_1$ = factorize_terminal($M_{1,2}$);
9 │ $symb_2$ = factorize_terminal($M_{1,2}$);
10 │ $symb_3$ = factorize_terminal(M_3);
11 $symbolic \leftarrow \emptyset$;
12 **foreach** $(\phi_i, m_i) \in symb_1$ **do**
13 **foreach** $(\phi_j, m_j) \in symb_2$ **do**
14 │ $m_{i,j} \leftarrow m_i \times m_j$;
15 │ **if** $(m_{i,j}, \phi) \in symbolic$ **then** update $symbolic[m_{i,j}] = \phi + \phi_i\phi_j$;
16 │ **else** set $symbolic[m_{i,j}] = \phi + \phi_i\phi_j$
17 **foreach** $(\phi_k, m_k) \in symb_3$ **do**
18 │ **if** $(m_k, \phi) \in symbolic$ **then** update $symbolic[m_k] = \phi + \phi_k$;
19 │ **else** set $symbolic[m_k] = \phi_k$;
20 **return** $symbolic$;

Algorithm 2 shows the details of the recursive symbolic factorization. First the monomials are filtered, like in the terminal factorization algorithm (line 1-2). In the following lines 3-6 the algorithm first checks whether to continue recursion or end with terminal factorizations. The recursion depth depends on the limit d and on the complexity of the monomials. In each recursion step the limit is reduced and $M_{1,2}$ is used for both p_1 and p_2. The monomials M_3 are used for the recursion of p_3. If the recursion condition does not hold, terminal factorizations are created instead (lines 8-10). Afterwards, in lines 11-19 the returned symbolic factorizations are merged to get the symbolic representation in the current level. This is done in a similar way as in the terminal algorithm. Instead of variables the expressions from the recursive calls are used.

E. Cost Function and Optimization

After creating the symbolic representation an area efficient factorization has to be found, i.e. a good valuation of the variables. A constraint solver satisfying Equation (5) creates a valid assignment, i.e. a new polynomial, which is possibly more expensive than $poly$ itself. Therefore the constraint solver has to have a notion of costs to be minimized. With respect to this cost function the most efficient factorization can be found. This section describes how to define a suitable cost function to reduce multiplications and enhance monomial reuse.

An intuitive cost function is the complexity of the used monomials, which indicates how many variables are multiplied to create the circuit. This is defined by extracting all terms from $poly_{sym}$ and sorting the coefficient variables by monomial m. Let $\Gamma_m = \{a | (a, m) \in T^L_{poly_{sym}}\}$ contain all coefficient variables a_i, b_j, c_k that form a term with the monomial m in $poly_{sym}$ (see Equation (3)). Under the assumption that a monomial has to be calculated once and can be reused multiple times, the costs for the monomial are added if any of its coefficients is non-zero:

$$cost_C := \sum_{m \in M^L_{poly_{sym}}} \begin{cases} C(m) & \text{if } \exists a \in \Gamma_m : a \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Additionally, for each symbolic partition one additional multiplier is required to create $p_1 \times p_2$. For this the predicate $\text{used}(p_i)$ is defined to be true iff any coefficient variable in p_i is not zero or any of the sub partitions is used. This enables the definition of $cost_\times$ as:

$$\begin{aligned} cost_\times(p_i) \quad := \quad & \text{ite}(\text{used}(p_{i,1}) \vee \text{used}(p_{i,2}), 1, 0) \\ + \quad & \text{ite}(\text{is terminal } p_i, 0, \\ & cost_\times(p_{i,1}) + cost_\times(p_{i,2}) + cost_\times(p_{i,3})) \end{aligned}$$

The overall cost of multiplications is given by

$$cost_{mult} = cost_C + cost_\times$$

Example 1: Given polynomial $p = x^2 - y^2$ containing the monomials and sub-monomials $x^2, x, y^2, y, 1$. The first partitioning step for $p = p_1 \times p_2 + p_3$ results in $M_{1,2} = \{x, y, 1\}$ and $M_3 = \{x^2, xy, y^2, x, y, 1\}$. The symbolic partitioning therefore results in:

$$\begin{aligned} p \quad = \quad & (a_1 x + a_2 y + a_3) \times (b_1 x + b_2 y + b_3) \\ & + (c_1 x^2 + c_2 xy + c_3 y^2 + c_4 x + c_5 y + c_6) \end{aligned}$$

From this form the constraints for factorization are derived. For each monomial in p the correct sum is constrained:

$$\begin{aligned} a_1 b_1 + c_1 \quad &= \quad 1 \quad (\text{for } x^2) \\ a_2 b_2 + c_3 \quad &= \quad -1 \quad (\text{for } y^2) \\ a_1 b_2 + a_2 b_1 + c_2 \quad &= \quad 0 \quad (\text{for } xy) \\ &\vdots \end{aligned}$$

The corresponding cost functions are

$$\begin{aligned} cost_C \quad := \quad & \text{ite}(c_1 \neq 0, 2, 0) \\ + \quad & \text{ite}(c_2 \neq 0, 2, 0) \\ + \quad & \text{ite}(c_3 \neq 0, 2, 0) \\ + \quad & \text{ite}(a_1 \neq 0 \vee b_1 \neq 0 \vee c_4 \neq 0, 1, 0) \\ + \quad & \text{ite}(a_2 \neq 0 \vee b_2 \neq 0 \vee c_5 \neq 0, 1, 0) \\ cost_\times \quad := \quad & \text{ite}(\bigvee_{i=1}^{3} a_i \neq 0 \vee \bigvee_{j=1}^{3} b_i \neq 0, 1, 0) \end{aligned}$$

The given polynomial $p = x^2 - y^2$ has costs of $cost_{mult}(p) = C(x^2) + C(y^2) = 4$. Using a constraint-solver and the constraint $cost_{mult}(poly_{sym}) \leq 4$, the solution $poly_{sym} = (x + y) \times (x - y)$ is found with minimal costs of 3.

After minimizing the number multipliers, an additional step is performed to minimize the number of adders. Similar to $cost_{mult}$, the number of additions in terminal partitions plus additions introduced through (recursive) partitioning is defined to be the cost of addition $cost_{add}$. Adder structures are used in terminal parts p_1, p_2, p_3 to sum up all terms. Therefore the number of terms n is counted and $n - 1$ adders are needed for p_i if at least two terms have non-zero coefficients:

$$\text{adders}(p) = \max\left(1, \sum_{(a,m)\in T_p} \text{ite}(a \neq 0, 1, 0)\right) - 1$$

Additionally for each symbolic partition one adder is used with $(p_1 \times p_2) + p_3$. The number of adders is therefore:

$$\begin{aligned} cost_+(p_i) \;:=\;& \text{ite}((\text{used}(p_{i,1}) \vee \text{used}(p_{i,2})) \wedge \text{used}(p_{i,3}), 1, 0) \\ +\;& \text{ite}(\text{is terminal } p_i \\ & , \text{adders}(p_{i,1}) + \text{adders}(p_{i,2}) + \text{adders}(p_{i,3}) \\ & , cost_+(p_{i,1}) + cost_+(p_{i,2}) + cost_+(p_{i,3})) \end{aligned}$$

We apply a two-step algorithm. The first step is to minimize the costs of multiplication. Afterwards $cost_{mult}$ is constrained to the optimal value. In the second step $cost_+$, i.e. the number of adders is minimized.

Other types of cost functions are also possible, e.g. a single-step optimization that considers adder and multiplier costs in a single cost function, e.g. by using the area required for each type. Furthermore modelling other targets as delay or common sub-polynomials is possible. On the other hand more complex cost function will typically increase the run-time.

As a constraint solver an (incremental) solver for *Satisfiability Modulo Theories* (SMT) for bit-vector theory [14] is used. Such a solver can naturally represent constraints over $\mathbb{Z}_{2^n}$ including modulo arithmetics and can efficiently find valid assignments or prove that no such assignment exists.

The factorization constraints are translated to SMT bit-vector logic with the symbolic coefficients as bit-vector variables. Furthermore the cost function is expressed in terms of these variables. The minimization algorithm uses multiple calls to the solver until the minimum is found. For this the algorithm assumes a cost value as upper bound, e.g. costs of input polynomial $poly$, and queries the solver whether a solution with costs less than the upper bound exists. In practice binary search using a lower and an upper bound was most efficient for finding the minimal cost value. The bounds are updated incrementally. If the constraint $cost < c$ is satisfiable, the upper bound is updated, otherwise the lower bound is updated. The value c in the next iteration is calculated as $c = \frac{upper+lower}{2}$. When $upper = lower$ holds, the minimum is found.

This algorithm always finds the minimal value of the cost function. Finding an optimal solution is expected to result in high run-times, therefore two restrictions were applied to reduce the size of the constraint solver instance:

- The factorization of p_3 is terminal for all cases.
- A more restrictive filter was applied that eliminates the top half of the monomials in each step.

This significantly decreases the run-time of the SMT solver, but excludes some possible factorizations.

V. Multi-Output Polynomial Optimization Algorithm

The single-output polynomial optimization algorithm can be extended to minimize the multipliers of several polynomials at the same time. This is based on the same assumption, that a monomial calculated once can be shared in multiple polynomials. This means that a monomial m used in $poly^{(1)}$ and $poly^{(2)}$ has to be synthesized only once.

The basic task is to find a good representation for a set of n polynomials $polys = \{poly^{(1)}, \ldots, poly^{(n)}\}$. The single-output polynomial optimization algorithm is extended to minimize such a set. The symbolic representation does not change, it is applied on each polynomial separately. For each polynomial $poly^{(i)}$ the symbolic representation $poly^{(i)}_{sym}$ is created according to Equation (3). Afterwards the respective factorization constraints as in Equation (5) are derived.

Only the cost function has to be adapted. The complexity costs are calculated over the monomials from all polynomials:

$$\begin{aligned} \Gamma'_m &= \{a | (a, m) \in \bigcup_{i=1}^{n} T^L_{poly^{(i)}_{sym}}\} \\ cost'_C &= \sum_{m \in \bigcup_{i=1}^{n} M^L_{poly^{(i)}_{sym}}} \begin{cases} C(m) & \text{if } \exists a \in \Gamma'_m : a \neq 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Hence, the overall cost of multiplications are now:

$$cost'_{mult} = cost'_C + \sum_{p \in polys} cost_\times(p)$$

To find an optimal assignment the same algorithm is applied as for the optimization of single-output polynomials. Note also that the multi-polynomial algorithm collapses to the single-polynomial algorithm when applied to only one polynomial. Considering the single polynomial is therefore only a special case of the general algorithm.

VI. Experimental Results

The experiments were run on an Intel Core 2 Duo CPU at 3.33 GHz with 3 GB of main memory running a Linux operating system. As SMT-Solver Boolector 1.1 was used [15]. In order to demonstrate the advantage of our proposed algorithms over the state-of-the-art techniques, we use several polynomials extracted from real embedded systems such as digital signal processing, computer graphics, automotive and communication applications. In the first experiment, we have employed phase-shift keying (PSK) that is used in digital communication from [4], digital image rejection unit (DIRU), Degree-5 Filter (PFLT), multivariate cosine wavelet polynomial (MVCS), an anti-aliasing function (ANTI) [4], a Savitzky Golay filter (SG2) and Quadratic polynomial (QUAD) benchmarks and applied our method to the benchmarks listed in Table I as single-output polynomials. These benchmarks have been used in previous work on polynomial datapath optimization such as [13], [10]. We have synthesized the polynomials using a traditional logic synthesis tool in 0.25μm CMOS technology.

The important parameters of the circuits including the area, counted by number of logic gates (Area) as well as the critical path delay in nanoseconds (Delay) are shown in Table I. Column M/V/D/n provides the number of monomials/the number of variables/the highest degree/the bit-vector size of each variable. In each experiment multiplier and adder minimization as described previously was applied.

We compare the results of our approach to the one of [13] that has been shown to perform about 20% to 40% better than state-of-the-art synthesis approaches. Even with the run-time optimizations described in Section IV, our algorithm takes considerably more time

Table I
COMPARISON OF THE FORMAL OPTIMIZATION TECHNIQUE WITH THE APPROACH IN [13] (SINGLE-OUTPUT POLYNOMIALS)

Benchmark	M/V/D/n	Technique in [13]			SMT-based		
		Time (s)	Delay (ns)	Area	Time (s)	Delay (ns)	Area
ANTI	7/1/6/16	<0.01	38.78	100 574	15.4	34.83	88 761
DIRU	8/2/4/16	3.10	23.17	55 631	57.9	20.57	48 501
MVCS	9/2/3/16	2.60	27.49	111 801	12,7	23.89	95 147
PFLT	6/1/5/16	<0.01	39.38	76 069	15.5	34.87	65 819
PSK	9/2/4/16	4.00	29.24	141 923	24.2	35.24	143 277
QUAD	5/2/2/16	<0.01	21.64	48 289	3.8	18.14	45 090
SG2	9/2/3/16	<0.01	29.21	149 496	6.4	25.58	127 578
average savings:						7.72%	10.47%

than [13]. This is not surprising, since we are comparing heuristic and exact algorithms. In all cases but one a significantly lower number of gates is achieved compared to [13]. As a side-effect the delay is also reduced on the same instances. The slightly poorer performance on PSK instance is due to the fact that the heuristic [13] is able to extract two sub-polynomials in the form of $p'_1 \times p'_2 + p'_3 \times p'_4 + p'_5$. This form is covered by recursive factorization of p_3 in $p_1 \times p_2 + p_3$ as $p'_1 \times p'_2 + (p'_3 \times p'_4 + p'_5)$. But as already described, recursive factorization of p_3 was disabled because it drastically impacts the run-time of the algorithm. On average a reduction of 10.47% in area and by 7.72% in delay is achieved.

In another experiment, we have employed different combinations of PSK, QUAD, MVCS, PFLT, DIRU, ANTI and SG2 benchmarks as multi-output polynomials. In addition to the results from [13] we also included results from [10]. Table II summarizes the results for 22 configurations. Our results show that for a given set of polynomial expressions, our approach determines better factorizations for all combinations but one with respect to the area. In instance 14, the heuristics outperform our algorithm. The technique in [13] is able to extract a common expression $(x + y)$ from multiple polynomials. Our algorithm cannot find this extraction as it does not result in an improvement of the cost function, i.e. the cost of multiplication does not decrease. On the other hand we are able to reduce the area on instance 20 by 30.93% and its delay by 15.17% compared to [13]. Furthermore on instance 5 we achieve a reduction of 46.14% in delay and 16.87% in area compared to [10]. Overall we obtained average savings of 12.15% in the area and 3.05% in the delay over [13]. Compared to [10] we achieved average savings of 21.17% in the delay and 5.72% in the area.

Table II
COMPARISON OF THE FORMAL OPTIMIZATION TECHNIQUE WITH THE APPROACHES IN [10] AND [13] (MULTI-OUTPUT POLYNOMIALS)

Number	Benchmark	M/V/D/n	Technique in [10]		Technique in [13]			SMT-based		
			Delay (ns)	Area	Time (s)	Delay (ns)	Area	Time (s)	Delay (ns)	Area
1	DIRU, PFLT	14/2/5/16	38,23	117 357	3.10	26.87	110 362	35.60	21.55	106 220
2	PSK, PFLT	15/2/5/16	31,75	172 597	4.00	31.03	193 366	18.10	31.99	184 223
3	QUAD, PFLT	11/2/5/16	38,29	158 975	0.01	30.88	123 192	18.20	39.07	112 102
4	ANTI, PFLT	13/1/6/16	54,30	143 097	0.02	39.92	154 170	35.80	40.30	131 284
5	MVCS, PFLT	15/2/5/16	37,91	159 732	2.60	27.89	156 335	46.10	20.42	132 791
6	DIRU, MVCS, PFLT	23/2/5/16	37,91	192 217	5.70	31.90	208 479	4 575.70	34.23	196 199
7	DIRU, PSK, PFLT	23/2/5/16	31,75	191 522	7.10	29.02	202 515	311.20	27.57	154 228
8	DIRU, QUAD, PFLT	19/2/5/16	38,34	190 864	3.10	29.88	176 156	120.10	26.05	152 455
9	MVCS, PSK, PFLT	24/2/5/16	31,75	215 592	6.60	31.05	279 886	30.70	32.04	240 726
10	MVCS, QUAD, PFLT	20/2/5/16	38,33	229 893	2.60	31.12	201 089	68.40	31.76	188 927
11	PSK, QUAD, PFLT	20/2/5/16	43,18	257 574	4.00	31.04	200 728	23.90	29.09	190 637
12	DIRU, ANTI, PFLT	21/2/6/16	54,38	178 268	3.10	39.93	209 439	208.70	40.06	154 724
13	DIRU, MVCS, PSK, PFLT	32/2/5/16	31,75	234 004	9.70	32.06	286 603	247.80	31.40	254 738
14	DIRU, MVCS, QUAD, PFLT	28/2/5/16	38,33	291 980	5.70	32.90	283 855	479.30	37.82	445 008
15	DIRU, PSK, QUAD, PFLT	28/2/5/16	43,07	249 654	7.10	30.02	248 303	12 512.20	31.74	221 264
16	MVCS, PSK, QUAD, PFLT	29/2/5/16	43,08	274 683	6.60	30.96	304 841	67.50	31.32	271 477
17	DIRU, MVCS, ANTI, PFLT	30/2/6/16	54,32	251 844	5.70	39.93	317 693	73.90	34.37	248 499
18	DIRU, MVCS, PSK, ANTI, PFLT	39/2/6/16	54,48	408 720	9.70	39.86	410 926	4 272.20	33.19	324 402
19	DIRU, MVCS, PSK, SG2, PFLT	41/2/5/16	41,71	339 997	9.70	39.56	458 484	6 167.30	35.56	326 957
20	DIRU, MVCS, PSK, QUAD, SG2, PFLT	46/2/5/16	41,71	389 744	9.70	39.56	498 052	6 289.30	33.56	343 990
21	DIRU, MVCS, PSK, SG2, ANTI, PFLT	48/2/6/16	54,59	394 574	9.70	39.86	529 409	759.10	36.94	428 977
22	DIRU, MVCS, PSK, QUAD, SG2, ANTI, PFLT	53/2/6/16	54,59	434 178	9.70	39.86	568 499	1 421.60	41.27	409 942
average savings:			21.17%	5.72%		3.05%	12.15%			

VII. CONCLUSIONS

This paper presented a methodology to describe polynomial factorization and minimization as a constraint solving problem. This methodology was extended to recursive factorization and multiple polynomials. The approach outperforms state-of-the-art optimization techniques. On average the algorithm improves area and delay for the benchmarks considered.

Future extensions of this approach are the realization of different cost functions such as delay as well as further run-time optimizations.

REFERENCES

[1] B. Alizadeh and M. Fujita, "Modular-HED: A canonical decision diagram for modular equivalence verification of polynomial functions," in *Proc. of 5th International Workshop on Constraints in Formal Verification (CFV)*, 2008, pp. 22–40.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *Journal of the ACM*, vol. 24, no. 1, pp. 146–160, 1977.

[3] M. A. Breuer, "Generation of optimal code for expressions via factorization," *Comm. of the ACM*, vol. 12, no. 6, pp. 333–340, 1969.

[4] A. Peymandoust and G. D. Micheli, "Application of symbolic computer algebra in high-level data-flow synthesis," *IEEE Trans. on CAD*, vol. 22, no. 9, pp. 1154–1165, 2003.

[5] S. Gopalakrishnan and P. Kalla, "Optimization of polynomial datapaths using finite ring algebra," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 4, p. 49, 2007.

[6] A. Hosangadi, F. Fallah, and R. Kastner, "Factoring and eliminating common subexpressions in polynomial expressions," in *Int'l Conf. on CAD*, 2004, pp. 169–174.

[7] B. DeRenzi and W. Gong, "JuanCSE," 2005, http://express.ece.ucsb.edu/suif/cse.html, last updated 2005. [Online]. Available: \url{http://express.ece.ucsb.edu/suif/cse.html}

[8] E. Kaltofen, J. P. May, Z. Yang, and L. Zhi, "Approximate factorization of multivariate polynomials using singular value decomposition," *J. Symb. Comput.*, vol. 43, no. 5, pp. 359–376, 2008.

[9] S. Gopalakrishnan, P. Kalla, M. B. Meredith, and F. Enescu, "Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors," in *Int'l Conf. on CAD*, 2007, pp. 143–148.

[10] S. Gopalakrishnan and P. Kalla, "Algebraic techniques to enhance common sub-expression elimination for polynomial system synthesis," in *Design, Automation and Test in Europe*, 2009, pp. 1452–1457.

[11] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *IEEE Trans. on CAD*, vol. 25, no. 10, pp. 2012–2022, 2006.

[12] O. Sarbishei, B. Alizadeh, and M. Fujita, "Polynomial datapath optimization using partitioning and compensation heuristics," in *Design Automation Conf.* ACM, 2009, pp. 931–936.

[13] B. Alizadeh and M. Fujita, "Improved heuristics for finite word-length polynomial datapath optimization," in *Int'l Conf. on CAD*. ACM, 2009, pp. 739–744.

[14] S. Ranise and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," http://www.smtlib.org, 2006.

[15] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.