Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis

Armita Peymandoust and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—The growing market of multimedia applications has required the development of complex application-specified integrated circuits with significant data-path portions. Unfortunately, most high-level synthesis tools and methods cannot automatically synthesize data paths such that complex arithmetic library blocks are intelligently used. Namely, most arithmetic-level optimizations are not supported and they are left to the designer's ingenuity. In this paper, we show how symbolic algebra can be used to construct arithmetic-level decomposition algorithms. We introduce our tool, SymSyn, that optimizes and maps data flow descriptions into data paths using complex arithmetic components. SymSyn uses two new algorithms to find either minimal component mapping or minimal critical path delay (CPD) mapping of the data flow. In this paper, we give an overview of the proposed algorithms. We also show how symbolic manipulations such as tree-height-reduction, factorization, expansion, and Horner transformation are incorporated in the preprocessing step. Such manipulations are used as guidelines in initial library element selection to accelerate the proposed algorithms. Furthermore, we demonstrate how substitution can be used for multiexpression component sharing and CPD optimization.**

***Index Terms*—Component mapping, data flow synthesis, design reuse, DSP synthesis, Gröbner basis, high-level synthesis, symbolic algebra.**

I. Introduction

AUTOMATING the design of data paths from high-level specifications is necessary to meet aggressive time-to-market requirements. The optimal choice of the arithmetic units implementing complex data flows strongly affects the cost, performance, and power consumption of the silicon implementations. Unfortunately, current commercial tools rely on synthesis directives (*pragmas*) from designers in order to map data flow into complex arithmetic library elements.

On the other hand, existing high-level synthesis tools are effective in capturing HDL models of the hardware and mapping them into control/data flow graphs (CDFGs), performing scheduling, resource sharing, retiming, and control synthesis [1]. The approach presented in this paper fits seamlessly into current high-level synthesis flow. We propose to analyze the data flow segments of the CDFG models in light of the arithmetic units available as library blocks, and to construct data paths that best exploit the given library. We assume that design is done using libraries that contain, beyond the basic elements such as adders and multipliers, more complex cells such as multiply/accumulate (MAC), sine, cosine, etc. An example of such a library is Synopsys DesignWare [2] library.

Two factors are key in automating the optimal mapping of data flow blocks. First, a functionality description formalism for data flow and library components. Second, a method supporting the decomposition of the data flow into a set of library elements. The functionality description formalism needs to be compact and canonical. Polynomial representation has been proven as an effective technique for representing both high-level specification and bit-level description of an implementation (library component) [3]–[5]. It has also been used in methods matching data flow clusters to library cells [3]–[5]. Unfortunately, such methods were limited to test for a match in the library of existing components. In case a match did not exist, there was no automated way to search for possible interconnections of library blocks matching the data flow cluster.

In this paper, we present our tool SymSyn that leverages results from Gröbner basis [9]–[12] applications and symbolic polynomial manipulation techniques to automate mapping of (a portion of) data flow into complex arithmetic library blocks. SymSyn framework contains two decomposition algorithms that assume the data flow and library elements are represented as polynomials. The first algorithm finds a minimal-component decomposition of a polynomial representing a (portion of) data flow. The decomposition is done in terms of arithmetic library elements, also represented as polynomials. Due to the importance of high-performance design, we have developed a second algorithm in the SymSyn framework to automatically map the data flow to arithmetic library elements such that the data flow has minimal critical path delay (CPD). The timing-driven decomposing algorithm uses various polynomial manipulation techniques as guidelines to achieve optimal component mapping and resource sharing for minimal delay.

As a motivating example, we consider the antialias function of an MP3 decoder that calculates the following equation in one of its basic blocks:

$$z = \frac{1}{2\sqrt{x^2+y^2}},$$

under the assumption that $x^2 + y^2 \geq \varepsilon > 0$.

A straightforward realization of this equation would use a divider and a square root operator, which are large and slow components and may not be available in the component library. For the sake of the example, we assume there are no square root and division operators available in our library. Alternatively, we

Manuscript received July 23, 2002; revised October 25, 2002. This work was supported in part by ARPA/MARCO Gigascale Research Center and in part by Synopsys Inc. This paper was recommended by Associate Editor R. Camposano.

The authors are with Stanford University, Computer Systems Laboratory, Stanford, CA 94305 USA (e-mail: armita@stanford.edu; nanni@stanford.edu).

Digital Object Identifier 10.1109/TCAD.2003.816213

0278-0070/03$17.00 © 2003 IEEE

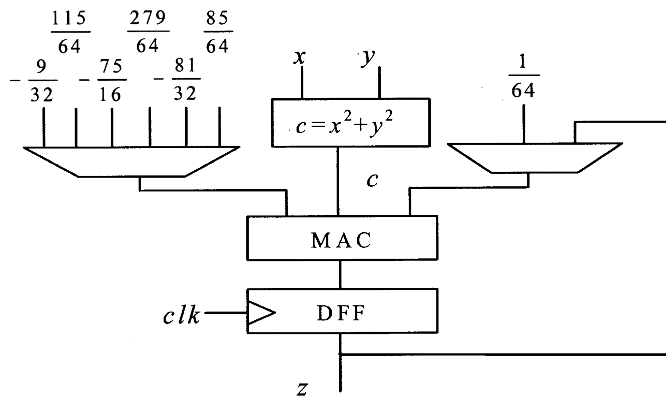

Fig. 1. Implementation for $\frac{1}{2\sqrt{x^2+y^2}}$.

assume the existence of an adder, multiplier, and multiplier-accumulator (MAC) in our library. Thus, $c = x^2+y^2$ can be easily computed. Next, using symbolic manipulations, we first substitute $x^2 + y^2$ by c and obtain

$$z = \frac{1}{2\sqrt{c}}.$$

We can approximate the given equation to a polynomial representation using the Taylor series expansion for a range of c based on the given application

$$z \cong \frac{1}{64}c^6 - \frac{9}{32}c^5 + \frac{115}{64}c^4 - \frac{75}{16}c^3 + \frac{279}{64}c^2 - \frac{81}{32}c + \frac{85}{64}.$$

The explanation is valid for a given range of c and the error can be computed using standard approximation methods [6]. If we perform a Horner-based transform on the polynomial approximation of z, we obtain

$$z \cong \frac{85}{64} + (-\frac{81}{32} + (\frac{279}{64} + (-\frac{75}{16} + (\frac{115}{64} + (-\frac{9}{32} + \frac{1}{64}c)c)c)c)c)c.$$

This formula can be implemented using a chain of six MACs, or one MAC in six cycles. Fig. 1 demonstrates one possible implementation.

Our synthesis tool, SymSyn, automates the algebraic manipulations shown in this example. SymSyn converts the basic blocks of a behavioral description, representing data flow portions of the design, to their polynomial representations and uses numerical methods for exact and inexact matching with library elements. If a match is not found, the data flow is decomposed into the library elements using symbolic computer algebra.

This paper is organized as follows. Section II gives an overview on symbolic algebra and explains how Gröbner basis is used in polynomial decomposition algorithms. In Section III, we present how we can leverage results from symbolic algebra to decompose a polynomial representing a (portion of) data flow. In Section III, we also explain our data flow synthesis tool, SymSyn, with an example. Sections IV and V describe the two new algorithms developed for automatic decomposition of data flow into complex arithmetic library components. Section VI shows a set of library independent symbolic transformations that are used to accelerate the proposed algorithms. Finally, Section VII explains the implementation of SymSyn and shows a set of experimental results.

II. Symbolic Computer Algebra

Traditional mathematical computation with computers and calculators is based on arithmetic of fixed-length integers and fixed-precision floating-point numbers, otherwise known as numeric computer algebra. In contrast, modern symbolic computation systems support exact rational arithmetic, arbitrary precision floating-point arithmetic, and algebraic manipulation of expressions containing undetermined values (symbols), such as variable `x` in `(x + 1)*(x - 1)`. Several commercial symbolic computer algebra systems are available on the market; Maple [7] and Mathematica [8] are most widely used.

The algebraic object that we would like to manipulate symbolically is a multivariate polynomial that represents a (portion of) data path of our design. We need to decompose this polynomial into polynomials representing the building blocks available in the target library. Such decomposition is called *simplification modulo set of polynomials* in symbolic computer algebra. Most symbolic polynomial manipulations that we find interesting in data-path synthesis are based on Gröbner bases [9]–[12]. Gröbner bases and Buchberger's algorithm generalize the division and greatest common divisor algorithms of univariate polynomials to multivariate polynomials. Therefore, it is the heart of symbolic polynomial factorization. Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of simplification modulo set of polynomials. In the following subsection, we will review Gröbner basis and its application to the *simplification* algorithm. Commercial symbolic computer programs, such as Maple [7], have a built-in routine that performs *simplification modulo set of polynomials*. In Maple this method is called *simplify*.

Next, we describe the underlying theory of *simplification modulo set of polynomials*. The reader solely interested in its application to data-path synthesis may proceed to Section II-C.

A. Basic Commutative Algebra

Definition 2.1: An *Abelian group* is a set G and a binary operation "$+$" satisfying all the following properties:

i) *Closure*. For every $\mathrm{a, b} \in \mathrm{G}$; $\mathrm{a + b} \in \mathrm{G}$.
ii) *Associativity*. For every $\mathrm{a, b, c} \in \mathrm{G}$; $\mathrm{a + (b + c) = (a + b) + c}$.
iii) *Commutativity*. For every $\mathrm{a, b} \in \mathrm{G}$; $\mathrm{a + b = b + a}$.
iv) *Identity*. There is an identity element $0 \in \mathrm{G}$ such that for all $\mathrm{a} \in \mathrm{G}$; $\mathrm{a} + 0 = \mathrm{a}$.
v) *Inverse*. If $\mathrm{a} \in \mathrm{G}$, then there is an element $\bar{\mathrm{a}} \in \mathrm{G}$ such that $\mathrm{a} + \bar{\mathrm{a}} = 0$.

Definition 2.2: A *commutative ring with unity* is a set R and two binary operations "$+$" and "$\cdot$", referred to as addition and multiplication, as well as two distinguished elements $0,\ 1 \in \mathrm{R}$ such that the following axioms hold:

i) R is an Abelian group with respect to addition with additive identity element 0;
ii) *Multiplication closure*. For every $\mathrm{a, b} \in \mathrm{R}$; $\mathrm{a \cdot b} \in \mathrm{R}$.

iii) *Multiplication associativity*. For every $\mathrm{a, b, c} \in \mathrm{R}$; $\mathrm{a \cdot (b \cdot c) = (a \cdot b) \cdot c}$.
iv) *Multiplication commutativity*. For every $\mathrm{a, b} \in \mathrm{R}$; $\mathrm{a \cdot b = b \cdot a}$.
v) *Multiplication identity*. There is a identity element $1 \in \mathrm{R}$ such that for all $\mathrm{a} \in \mathrm{R}$; $\mathrm{a} \cdot 1 = \mathrm{a}$.
vi) *Distributivity*. For every $\mathrm{a, b, c} \in \mathrm{R}$; $\mathrm{a \cdot (b+c) = a \cdot b + a \cdot c}$ holds for all $\mathrm{a, b, c} \in \mathrm{R}$.

Definition 2.3: A *field* K is a commutative ring with unity, where every element in K except 0 has a multiplicative inverse, i.e, $\forall \mathrm{a} \in \mathrm{K} - \{0\}$, $\exists \hat{\mathrm{a}} \in \mathrm{K}$ such that $\mathrm{a} \cdot \hat{\mathrm{a}} = 1$.

The set of all multivariate polynomials with variables $x_1, x_2, \ldots, x_n$, coefficients from a field K, and the two operations addition and multiplication forms a commutative ring with unity denoted by $\mathrm{R}[x_1, x_2, \ldots, x_n]$.

Definition 2.4: Let R be a commutative ring, a nonempty subset $\mathrm{I} \subseteq \mathrm{R}$ is an ***ideal*** when

i) $0 \in \mathrm{I}$;
ii) $\mathrm{p + q} \in \mathrm{I}$ for all $\mathrm{p, q} \in \mathrm{I}$;
iii) $\mathrm{r \cdot p} \in \mathrm{I}$ for all $\mathrm{p} \in \mathrm{I}$ and $\mathrm{r} \in \mathrm{R}$ [12].

Lemma 2.1: Let $P = \{p_1, p_2, \ldots, p_k\}$ be a finite subset of the polynomial ring $\mathrm{R}[x_1, x_2, \ldots, x_n]$ and $\langle P \rangle = \langle p_1, p_2, \ldots, p_k \rangle = \{\sum_{i=1}^{k} h_i \cdot p_i \,|\, h_i \in \mathrm{R}[x_1, x_2, \ldots, x_n]\}$. Then $\langle \mathrm{P} \rangle$ is an ideal in $\mathrm{R}[x_1, x_2, \ldots, x_n]$. We will call $\langle P \rangle$ the ideal generated by P and the set P is called generator or ***basis*** of this ideal. For example, the set of polynomials $P = \{p_1, p_2, p_3\}$ defined below generates a polynomial ideal over $\mathrm{R}[x_1, x_2, x_3]$

$$\begin{aligned} p_1 &= x_1^3 x_2 x_3 - x_1 x_3^2, \; p_2 = x_1 x_2^2 x_3 - x_1 x_2 x_3, \\ p_3 &= x_1^2 x_2^2 - x_3^2 \\ \langle P \rangle &= \{a_1 \cdot p_1 + a_2 \cdot p_2 + a_3 \cdot p_3 \mid a_1, a_2, a_3 \in \mathrm{R}[x_1, x_2, x_3]\}. \end{aligned}$$

Unfortunately, while P generates the infinite set $\langle P \rangle$, the polynomials p_i in P may not yield much insight into this ideal, since, for each ideal in a polynomial ring, there are many possible sets of polynomials that generate the ideal. In other words, the ideal basis is not unique. However, Buchberger [9] has shown that an arbitrary ideal basis can be transformed into a basis with special properties, which is called the *Gröbner basis*. A minimal (or reduced) Gröbner basis forms a canonical representation for a multivariate polynomial ideal. A canonical representation for ideals enables us to check whether two ideals are equal. Important applications of the Gröbner basis include polynomial decomposition and variable elimination in a set of multivariate polynomials. One may say that the Gröbner basis is the cornerstone of polynomial decomposition used in our mapping algorithm. In the next section, we will give a brief description of Buchberger's algorithm.

B. Gröbner Bases

Before introducing a formal definition of Gröbner bases, we need to define *term ordering* and *reduction* (division) of multivariate polynomials. A monomial of the form $x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$, where $x_1, x_2, \ldots, x_n$ are the variables of the polynomial and $(i_1, i_2, \ldots, i_n) \in Z_{\geq 0}^n$ are the exponents, is called a ***term***. We denote the set of terms of the polynomial ring $\mathrm{R}[x_1, x_2, \ldots, x_n]$ by $\mathrm{T}_{\mathbf{x}}$, where $\mathbf{N}$ is the set of nonnegative integers

$$\mathrm{T}_{\mathbf{x}} = \{x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} \mid i_1, i_2, \ldots, i_n \in \mathbf{N}\}.$$

In division of univariate polynomials, R[x], we start by writing the polynomials such that its terms are in decreasing order of the degree of x. To define reduction (division) for multivariate polynomials, we need to have an ordering for multivariate term.

Definition 2.5: A *term ordering* on $\mathrm{R}[x_1, x_2, \ldots, x_n]$ is any relation $>$ on $Z_{\geq 0}^n$ satisfying the following.

i) $>$ is a total (or linear) on $Z_{\geq 0}^n$.
ii) If α, β, and $\gamma \in Z_{\geq 0}^n$ and $\alpha > \beta$, then $\alpha + \gamma > \beta + \gamma$.
iii) $>$ is well ordered on $Z_{\geq 0}^n$. This means that every nonempty subset of $Z_{\geq 0}^n$ has a smallest element under $>$.

The *leading monomial* of polynomial $p \in \mathrm{R}[x_1, x_2, \ldots, x_n]$ with respect to a total ordering of the variables, such as the lexicographical ordering, is the monomial in p whose term is the maximal among those in p; we denote this monomial by $\mathrm{M}(p)$. We also define $\mathrm{hterm}(p)$ to be the maximal term, and the $\mathrm{hcoeff}(p)$ to be the corresponding coefficient, therefore,

$$\mathrm{M}(p) = \mathrm{hcoeff}(p) \cdot \mathrm{hterm}(p).$$

Example: Consider $p \in \mathrm{R}[x_1, x_2]$ that is written in lexicographical order

$$\begin{aligned} p &= 3x_1^2 x_2 + 5x_1^2 + x_2^2 \\ \mathrm{M}(p) &= 3x_1^2 x_2 \\ \mathrm{hterm}(p) &= x_1^2 x_2 \\ \mathrm{hcoeff}(p) &= 3. \end{aligned}$$

Definition 2.6. Reduction: For nonzero $p, q \in \mathrm{R}[x_1, x_2, \ldots, x_n]$ we say that p reduces modulo q if there exists a monomial in p which is divisible by $\mathrm{hterm}(q)$. Let $\alpha \in \mathrm{R}[x_1, x_2, \ldots, x_n] - \{0\}$, i.e., the ring of polynomials after removing the trivial 0 polynomial. If $p = \alpha \cdot t + r$, where $t \in \mathrm{T}_{\mathbf{x}}$, $r \in \mathrm{R}[x_1, x_2, \ldots, x_n]$, and $u = t/\mathrm{hterm}(q)$, $u \in \mathrm{T}_{\mathbf{x}}$, then we write $p \rightarrow_q p'$ to signify that p reduces to p' (modulo q) and p' is equal to

$$p' = p - \frac{\alpha \cdot t}{\mathrm{M}(q)} \cdot q = p - \frac{\alpha}{\mathrm{hcoeff}(q)} u \cdot q.$$

Example: Consider the following two polynomials:

$$\begin{aligned} p &= 6x^4 + 13x^3 - 6x + 1 \\ q &= 3x^2 + 5x - 1 \\ p &\rightarrow_q p' \qquad p' = p - 2x^2 \cdot q = 3x^3 + 2x^2 - 6x + 1. \end{aligned}$$

If p reduces to p' modulo a polynomial in a set of polynomials $Q = \{q_1, q_2, \ldots, q_n\}$, we say that p reduces modulo Q and write $p \rightarrow_Q p'$ ($p' = \mathrm{Reduce}(p, Q)$); otherwise, we say that

p is irreducible modulo Q. We denote, $p \to^{+}{}_{Q} p'$ if and only if there is a sequence such that

$$p = p_0 \to_Q p_1 \to_Q \ldots \to_Q p_n = p'.$$

If $p \to^{+}{}_{Q} q$ and q is irreducible, we will write $p \to_Q^* q$. It can be shown that for a fixed set Q and a given term ordering, the sequence of reductions is finite [10]. Therefore, we can construct Algorithm 2.1 which, given a polynomial p and set Q, finds a polynomial q such that $p \to^{*}{}_{Q} q$. In Algorithm 2.1, $R_{p,Q}$ denotes the set polynomials in $Q - \{0\}$ such that $\mathrm{hterm}(p)$ is divisible by $\mathrm{hterm}(q)$. Note that any member of $R_{p,Q}$ can be chosen in each iteration, but this choice affects the efficiency of the algorithm. For the sake of simplicity, we assume an efficient selection is implemented in *selectpoly*.

As mentioned previously, any finite set of polynomials Q generates an ideal $\langle Q \rangle$ and Q is called the basis of this ideal. If a nonzero polynomial p is reduced to zero modulo Q, we can determine that p is a member of the ideal generated by Q: $p \to^{*}{}_{Q} 0 \Rightarrow p \in \langle Q \rangle$. However, the converse is not true for all basis of $\langle Q \rangle$.

Algorithm 2.1 Full Reduction of p Modulo Q.
procedure $\mathrm{Reduce}(p, Q)$
  # Given a polynomial p and a set of polynomials Q
  # from the ring $\mathrm{R}[x_1, x_2, \ldots, x_{nz}]$, find a q such that $p \to^{*}{}_{Q} q$.
  # Start with the whole polynomial.
  $r \leftarrow p; q \leftarrow 0$
  # if no reducers exist, strip off the leading
  # monomial; otherwise, continue to reduce.
  while $r \neq 0$ **do** {
    $\mathrm{R} \leftarrow \mathrm{R}_{r,Q}$
    while $\mathrm{R} \neq \emptyset$ **do** {
      #select a polynomial $\in \mathrm{R}$
      $f \leftarrow selectpoly(\mathrm{R})$
      $\mathrm{R} \leftarrow \mathrm{R} - \{f\}$
      $r \leftarrow r - (\mathrm{M}(r)/\mathrm{M}(f))f$
    }
    $q \leftarrow q + \mathrm{M}(r); r \leftarrow r - \mathrm{M}(r)$
  }
  $\mathbf{return}(q)$
end

Definition 2.7: An ideal basis $G \subset \mathrm{R}[x_1, x_2, \ldots, x_n]$ is called a *Gröbner basis* (with respect to a fixed term ordering and the implied permutation of variables) when $p \to^{*}{}_{G} 0 \Leftrightarrow p \in \langle G \rangle$.

We define the *S-polynomial* of $p, q \in \mathrm{R}[x_1, x_2, \ldots, x_n]$, denoted as $\mathrm{Spoly}(p, q)$, as

$$\mathrm{Spoly}(p, q) = \mathrm{LCM}\big(\mathrm{M}(p), \mathrm{M}(q)\big) \cdot \left[\frac{p}{\mathrm{M}(p)} - \frac{q}{\mathrm{M}(q)}\right].$$

Example: For polynomials $p = 3x^2y - y^3 - 6$ and $q = 6xy^3 + 5x - 1$ with degree ordering we have $\mathrm{LCM}(\mathrm{M}(p), \mathrm{M}(q)) = \mathrm{LCM}(3x^2y, 6xy^3) = 6x^2y^3$

$$\begin{aligned}\mathrm{Spoly}(p, q) =& 6x^2y^3 \cdot \left[\frac{3x^2y - y^3 - 6}{3x^2y} - \frac{6xy^3 + 5x - 1}{6xy^3}\right]\\ =& -2\mathrm{y}^5 - 12\mathrm{y}^2 - 5\mathrm{x}^2 + \mathrm{x}.\end{aligned}$$

Algorithm 2.2 Buchberger's Algorithm for Gröbner Bases.
procedure $\mathrm{Gbasis}(Q)$
  # Given a set of polynomials Q, compute G such that $\langle G \rangle = \langle Q \rangle$ and G is a Gröbner basis.
  $G \leftarrow Q; k \leftarrow \mathrm{length}(G)$
  # Initialize B to all possible pairs
  $B \leftarrow \{[i, j] : 1 \leq i < j \leq k\}$
  while $B \neq \emptyset$ **do** {
    $[i, j] \leftarrow$ select a pair from B
    # mark that pair as selected
    $B \leftarrow B - \{[i, j]\}$
    # G_i denotes the ith element of the ordered set G
    $h \leftarrow \mathrm{Reduce}(\mathrm{Spoly}(G_i, G_j), G)$
    if $h \neq 0$ **then** {
      $G \leftarrow G \cup \{h\}; k \leftarrow k + 1$
      $B \leftarrow B \cup \{(i, k) : 1 \leq i < k\}\}\}$
    $\mathbf{return}(G)$
end

In can be shown that [6], [9], G is a Gröbner basis when:

1) the only irreducible polynomial in $\langle G \rangle$ is $p = 0$;
2) $\mathrm{Spoly}(p, q) \to_G^+ 0$ for all $p, q \in G$;
3) if $p \to_G^* q$ and $p \to_G^* r$, then $q = r$.

Buchberger's algorithm (Algorithm 2.2) uses the properties above to convert a finite set $Q \subset \mathrm{R}[x_1, x_2, \ldots, x_n]$ into a Gröbner basis [9].

In order to check whether a polynomial p is a member of the ideal $\langle Q \rangle$, first Algorithm 2.2 is used to form G a Gröbner basis for $\langle Q \rangle$. Procedure $\mathrm{Reduce}(p, G)$ (Algorithm 2.1) must then return zero.

C. Gröbner Bases and Data-Path Synthesis

We now describe the application of the theory described previously. Let L be the set of polynomial representations of the library elements. In order to synthesize a data path for a polynomial representation S using library L, S should be a member of $\langle L \rangle$. In order to examine membership in $\langle L \rangle$, we need to calculate G the Gröbner basis of $\langle L \rangle$ and use $\mathrm{Reduce}(S, G)$. If S reduces to zero, then $S \in \langle L \rangle$. If S is reduced to zero only using polynomials in G that are also in L, then S can be built from the given library elements. As an example, consider

$$\begin{aligned}\mathrm{S} =& \mathrm{x} + \mathrm{x}^2 + \mathrm{x}^3 + \mathrm{y} + \mathrm{xy} + \mathrm{x}^2\mathrm{y}\\ \mathrm{L} =& \{1 + \mathrm{x} + \mathrm{x}^2,\ \mathrm{x} + \mathrm{y}\}\\ G =& \mathrm{Gbasis}(L) = \{\mathrm{x} + \mathrm{y},\ \mathrm{y}^2 - \mathrm{y} + 1\}\end{aligned}$$
$$\mathrm{Reduce}(S, G) \text{ returns zero, therefore, } S \in \langle L \rangle.$$

While performing $\mathrm{Reduce}(S, G)$, we determine that

$$S = (\mathrm{x} + \mathrm{y})(1 + \mathrm{x} + \mathrm{x}^2)$$
$$\text{therefore, } S \text{ can be decomposed into elements of } \langle L \rangle.$$

III. Symbolic Algebra and Library Matching

After extracting the CDFG of an algorithmic level DSP model, we calculate the polynomial representations of its basic blocks. The polynomial representation of a basic block can be directly extracted from algorithmic-level code if the basic

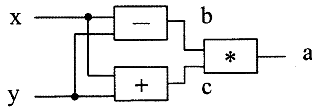

Fig. 2. Implementation of $x^2 - y^2$.

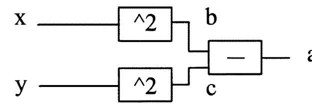

Fig. 3. Another implementation of $x^2 - y^2$.

block calculates a polynomial function. If the basic block performs a series of bit manipulations or Boolean functions, interpolation-based algorithms [3] can be used to formulate the equivalent polynomial representation. When the basic block implements a transcendental function, we use an approximation, such as the Taylor or Chebyshev series expansion, as its polynomial. The chosen polynomial approximation has to be verified manually by simulation to ensure that constraints, such as accuracy, are satisfied.

Symbolic computer algebra is subsequently used to intelligently decompose data flow to library components and automatically synthesize the data path. The symbolic algebra routine used in this algorithm is *simplification modulo set of polynomials* that has been described in Section II. Assume a basic block (or part of it) is represented by polynomial p and the library components available are represented by a set of polynomials L. As a reminder, to simplify a polynomial p modulo the side relation set L, we build a Gröbner basis from L, $G \leftarrow \text{Gbasis}(L)$, and use $\text{Reduce}(p, G)$ to obtain the simplified answer. The built-in function that implements *simplification modulo set of polynomials* in Maple is called *simplify* [7]. In order to comply with Maple terminology, we call the set of polynomials the *side relations*.

Note that any polynomial representation can be implemented using only adders and multipliers. Therefore, any polynomial representation of a basic block is guaranteed an implementation if the library includes adder and multiplier. Our goal is to find nontrivial solutions that are minimal in terms of component count or CPD. As an example, consider a data flow implementing $x\hat{}2 - y\hat{}2$ and a library that includes add, multiply subtract and square functions. Using Maple syntax we have the following.

$$> \mathtt{a} := \mathtt{x}\hat{}2 - \mathtt{y}\hat{}2 : \mathtt{siderels} := \{\mathtt{b} = \mathtt{x} - \mathtt{y},\ \mathtt{c} = \mathtt{x} + \mathtt{y}\};$$
$$> \mathtt{simplify}(\mathtt{a}, \mathtt{siderels}, [\mathtt{x}, \mathtt{y}, \mathtt{b}, \mathtt{c}]);$$
$$\mathtt{b}*\mathtt{c}$$

This is equivalent to the implementation shown in Fig. 2. Note that siderels is a subset of our library. Maple computes the Gröbner basis G of siderels and prints out the result of $\text{Reduce}(\mathtt{a}, \mathtt{siderels})$. The result indicates that

$$\mathtt{a} := \mathtt{x}\hat{}2 - \mathtt{y}\hat{}2 := \mathtt{b}*\mathtt{c} := (\mathtt{x} - \mathtt{y})*(\mathtt{x} + \mathtt{y}).$$

If the side relation set is changed, other possible solutions for the specification might be found.

$$> \mathtt{a} := \mathtt{x}\hat{}2 - \mathtt{y}\hat{}2 : \mathtt{siderels} := \{\mathtt{b} = \mathtt{x}\hat{}2,\ \mathtt{c} = \mathtt{y}\hat{}2\};$$
$$> \mathtt{simplify}(\mathtt{a}, \mathtt{siderels}, [\mathtt{x}, \mathtt{y}, \mathtt{b}, \mathtt{c}]);$$
$$\mathtt{b} - \mathtt{c}$$

results in the implementation shown in Fig. 3.

As shown, different side relation sets can result in different implementation of the specification. Therefore, to find the best possible implementation, the side relation set should be set equal to all subsets of the library with all possible permutations of the input variables. Since this is exponentially expensive, a guided architectural exploration is necessary. In the next two sections, we will introduce two algorithms designed to reduce the complexity of this search with two different final objectives. The first algorithm finds the minimal component decomposition for the given data flow. The second algorithm finds the minimal CPD implementation of the data flow.

IV. Minimal Component Decomposition Algorithm

In this section, we introduce one of the algorithms implemented in our tool, SymSyn. This algorithm automatically maps a polynomial representation of a (portion of) data flow to a set of complex arithmetic library components while using the least number of library components. This algorithm, in conjunction with classical high-level synthesis algorithms, can be used for efficient high-level DSP synthesis. The minimal component decomposition algorithm described is empowered by Gröbner basis fundamentals, described in Section II. The inputs to this algorithm are polynomial representation of the data flow basic block to be synthesized and a set of polynomials that represent the set of complex arithmetic library components available to the designer. As mentioned in the previous section, different side relation sets result in different implementations of the data flow. Therefore, our algorithm aims at intelligent side relation set selection to accelerate the decomposition process for a given criteria. The high-level view of the selection criteria for a minimal number of components is illustrated in Algorithm 4.1.

Let S be the polynomial representation of the basic block to be decomposed into complex library elements. We start by simplifying S modulo each library element as the side relation. The simplification results are stored in a tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the simplification result stored in a tree node does not correspond to a library element, we recursively apply the same steps to the new tree node.

Algorithm 4.1 Decompose S into elements of library L
procedure $\text{Decompose}(S, L)$
 # Given a polynomial representation of the spec S and a set of polynomials L as component library,
 # decompose S into elements of library L.
 # initialize tree
 $\text{treeroot}(S)$;
 $depth \leftarrow 0$
 $bound \leftarrow -1$
 while depth $\neq$ bound **do** {
  $bound \leftarrow$ $\mathbf{Explore}(S, L, depth)$ # Explore is defined below
  $depth \leftarrow depth + 1$

```
  }
 report best solution in tree
end
# used in Decompose procedure
int function Explore(S, L, d)
 bound ← −1
  for all n ∈ in tree with depth d do {
   for all sr ∈ L do {
    result = simplify(n, sr);
    # make result a child of node n
    addchild(n, result);
    if result ∈ L
      # solution is found
     bound = treedepth(result); }}
  # returns − 1 if no solution is found yet.
  return(bound)
end
```

To further reduce the search space a bounding function is used. The bounding function is the number of library components used to build the specification. In other words, if we find a solution with two library components we will not explore solutions requiring more than two components. But we will uncover all two-component solutions and choose the one with optimal cost (area or delay). The number of components used is the same as the depth of the simplification tree; therefore, the tree is bounded by the depth of the first solution found.

Such bounding function is chosen assuming that if a component is custom designed to perform a combination of arithmetic operations, it is more cost effective than connecting a series of components that perform the same arithmetic operations. Clearly, the merit of the result is strongly dependent on the available library.

A. Minimal Component Example

To clarify the algorithm described above, we choose our library to be a subset of the Synopsys DesignWare library consisting of six combinational elements: multiplier, adder, subtracter, multiplier-accumulator, sine, and cosine. As an example, consider synthesizing a phase shift keying (PSK) modulator used in digital communication. A data flow basic block of PSK has the following polynomial representation:

```
> S := 1 − .5*x0^2 − x0*x1 − .5*x1^2 + .041 667*x0^4
  + .166 668*x0^3*x1 + .250 002*x0^2*x1^2
  + .166 668*x0*x1^3 + .041 667*x1^4;
```

As the first step, SymSyn initializes a tree data structure and stores polynomial S in the root of the tree. For all library elements, SymSyn makes a call to Maple and performs *simplify* with side relation set equal to the library element. The results reported by Maple are kept as new children of the S tree node.

In the first iteration of our example, the side relation is set to the first element in the library, the multiplier. Shown below are the Maple commands. The first two lines are the requests sent by SymSyn, and the third line is the simplification result reported by Maple to SymSyn. SymSyn searches for a component in the library that implements the result, but it is not successful to find one for this instance.

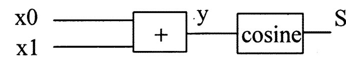

Fig. 4. Mapping S to two components.

```
> siderel := {y = x0*x1};
> simplify(S, siderel, [x0, x1, y]);
.041 667*x0^4+.166 668*x0^2*y−.5*x0^2+.041 667*x1^4
  +.166 668*x1^2*y − .5*x1^2 + .250 002*y^2 − 1.*y + 1
```

In the second iteration, the same steps are performed with the adder as the side relation. The simplification result now matches an approximation to the cosine function. Therefore, SymSyn marks this node as one possible solution. The following Maple commands show the result of this iteration. Note that the result is a Taylor series approximation of cosine. Since cosine is one of our library elements, we have found one possible solution, shown in Fig. 4.

```
> siderel := {y = x0 + x1};
> simplify(S, siderel, [x0, x1, y]);
  1. + .041 667*y^4 − .5*y^2
```

Since there is a solution with depth equal to *one* in the tree, a bound of *one* is set on the tree growth. Note that the root is denoted with depth equal to *zero*. Therefore, a solution at depth *one* consists of two components. SymSyn performs the steps described above for the rest of library elements and keeps the results in root offsprings. After going through all library elements, SymSyn finds only one solution using two components. The solution is demonstrated in Fig. 4. SymSyn will stop decomposing the leaf nodes, since continuation would result in a search for solutions with three or more components while the objective is to find a solution using minimal number of components.

V. Timing-Driven Decomposition Algorithm

In this section, we introduce the second algorithm implemented in our tool SymSyn. In contrast to the algorithm described in Section IV, here, we focus on minimizing the CPD of the data flow implementation. Previously, we focused on minimizing the number of components used to implement the data flow. Similar to Algorithm 4.1, this algorithm selects side relation sets intelligently to accelerate the decomposition process, since selecting different side relation sets result in different implementations of the data flow.

After extracting the CDFG of an algorithmic-level DSP model, the polynomial representations of its data flow basic blocks are passed as inputs to the timing-driven decomposition algorithm. Algorithm 5.1 shows the pseudocode of the timing-driven decomposition algorithm. This algorithm takes the same inputs as Algorithm 4.1; the polynomial representation of the basic block to be implemented and the polynomial representations of the complex library elements. Algorithm 5.1, uses the branch-and-bound method to reduce the side-relation-set selection space while searching for the implementation

with the least CPD. We define the bounding function as the best CPD of implementations seen so far. The lower bound computed at each decision branch is the CPD of components in the side relation set in view of data dependencies. If this lower bound is greater than the best CPD of implementations seen so far, the corresponding decision branch is pruned.

Algorithm 5.1 Decompose S into elements of library L
function GuidedDecomposition(*exp_tree, max_CPD, L*) {
  # initialize a solution tree
  $solution_tree \leftarrow \text{tree}(exp_tree)$;
  $depth \leftarrow 0$
  $bound \leftarrow \text{max_}CPD$
  for all $n \in$ in $solution_tree$ with depth $== depth$ **do** {
    if $depth == 0$ **then**
      choose all $sr \in L$ that preserve the *exp_tree* structure
    else for all $sr \in L$ **do** {
      if cost of $sr +$ cost of node $n < bound$ **then** {
      $result = \text{simplify}(n, sr)$;
      # make *result* a child of node n
      $\text{addchild}(n, result)$;
      add cost of *sr* to cost of *result*;
      if $result \in L$ **then** {
        # solution is found
       $bound =$ cost of node $result$; }
      if no more $n \in$ in $solution_tree$ with depth $== depth$
       $depth \leftarrow depth + 1$
    }}
  return the best solution in *solution_tree*
end
int function CalcMaxCPD(*expression_tree*) {
  CPD = the critical path delay of *expression_tree* assuming the expression is mapped to adders and multipliers only.
  return(*CPD*)
end
procedure $\text{main}(S, L)$
Given a polynomial representation of the spec S and a set of polynomials L as component library,
decompose S into elements of library L such that the CPD of S is minimized.
perform expression manipulation techniques

$exp_tree[1..\text{NumberOfManipulations}]$
$= \textbf{AllManipulations}(\text{S})$;

  for $i = 1$ **to** NumberOfManipulations **do** {
   $\text{max}CPD[i] = \textbf{CalcMaxCPD}(exp_tree[i])$;

$solution[i]$
   $= \textbf{GuidedDecomposition}(exp_tree[i], \text{max}CPD[i])$;

  }
  report the best solution in $solutions[i]$
end

Let S be the polynomial representation of the data flow. Our goal is to decompose S into the elements of the library L such that the CPD of S is minimized. Decomposing S is synonymous to simplifying S modulo elements of the library L as side relations. In order to decide which library elements should be used as the side relations, we use a decision tree (*solution_tree*) to implement the branch-and-bound algorithm. The bounding variable is initialized to the CPD of mapping the polynomial solely to adders and multipliers, also known as the lexicographical mapping.

The *simplify* results are also saved in the tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the CPD of the solution is smaller than previously encountered solutions, we set the bounding variable to the current delay. In case the simplification result stored in a tree node does not correspond to any library elements, we recursively apply the same steps to the new tree node.

In general, the branch-and-bound algorithm is practically applicable to most problems. However, introducing heuristics that lead quickly to promising solutions can improve the execution time without hampering the quality of the solution. As for all branch-and-bound algorithms, the worst case complexity remains exponential.

We use the expression manipulation techniques presented subsequently in Section VI as heuristic guidelines for choosing the side relation set. Initially, we apply tree-height reduction (THR), factorization, expansion, and Horner-based transform on S. As a result, we have several polynomials (*exp_tree*) representing the same data flow. Each of these representations can result in the desirable implementation based on the available library elements. Starting with the primary inputs, we try covering the expression tree with the library elements. We choose all library elements that cover the primary inputs and a portion of the expression tree as a side relation. If the result of simplify modulo side relation is not a library element, we decompose the result without further guidance from the expression tree. Algorithm 5.1 in conjunction with substitution and THR can be generalized to several polynomials in a basic block or across basic blocks.

A. Timing-Driven Example

As an example, consider a data flow segment of a Gabor filter with the following polynomial representation:

$$S = 1 - a^2 - b^2 + a^2b^2 + \frac{1}{2}a^4 + \frac{1}{2}b^4 - \frac{1}{6}a^6 - \frac{1}{2}a^4b^2 - \frac{1}{2}a^2b^4 - \frac{1}{6}b^6 + \frac{1}{24}a^8 + \frac{1}{6}a^6b^2 + \frac{1}{4}a^4b^4 + \frac{1}{6}a^2b^6 + \frac{1}{24}b^8.$$

Assume we would like to map S to a library consisting of functions implementing add, multiply, MAC, square, exp. After factorization, S will be converted to:

$$S = \frac{1}{24}(a^2 + b^2)(a^6 - 4a^4 + 3a^4b^2 + 12a^2 - 8a^2b^2 + 3a^2b^4 + 12b^2 + 24 - 4b^4 + b^6) + 1.$$

The factored form of S guides us to use `c = a^2 + b^2` as an initial side relation and sets an initial bound by mapping the fac-

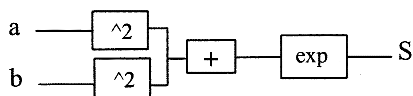

Fig. 5. Mapping S to four components.

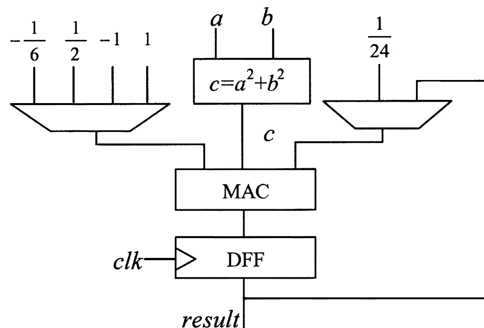

Fig. 6. Possible implementation for e^c.

tored form lexicographically to adders and multiplier. SymSyn makes a call to Maple and requests result of the following simplifying operation.

```
> siderel := c = a^2+b^2;
> result := simplify(S, siderel, [a, b, c]);
```

$$\texttt{result}=1-c+\frac{1}{2}*c\wedge 2-\frac{1}{6}*c\wedge 3+\frac{1}{24}*c\wedge 4$$

The last line is the result reported to SymSyn by Maple. As it can be seen, the result is a Taylor series expansion of $\exp(c)$. Therefore, the data flow can be implemented using two square components, an adder, and one exp component, as shown in Fig. 5. The bounding function is now changed to the CPD of the potential implementation. By exploring the other branches of the decision tree (*solution_tree*), we realize that all other branches are pruned by the new bound. Therefore, Fig. 5 is implementation with the least CPD.

Now, assume that there is no exp block in our library. In order to show the power of other polynomial transformations, we perform Horner transform (see Section VI-C) on the polynomial *result*

$$\text{result} = 1 + \left(-1 + \left(\frac{1}{2} + \left(-\frac{1}{6} + \frac{1}{24}\cdot c\right)\cdot c\right)\cdot c\right)\cdot c.$$

The formula given above can be implemented using a chain of four MACs or one MAC in four cycles. Fig. 6 demonstrates one possible implementation.

VI. Expression Manipulation Techniques

In Section V, an algorithm was introduced that maps a polynomial representation of a (portion of) data flow to complex arithmetic library elements such that the CPD is minimized. This algorithm was implemented in the Symbolic Synthesis tool, SymSyn. To accelerate the speed of minimal CPD decomposition in SymSyn, a guideline is necessary for side-relation selection. Such guideline should facilitate mapping for maximum parallelism. We have chosen different symbolic polynomial manipulation techniques as such guidelines. These transformations are the counterparts of the library independent transformations used in logic synthesis [1]. These heuristics can also be used as an enhancement to the minimal component decomposition algorithm. The intent of this section is to describe the manipulation techniques through simple examples.

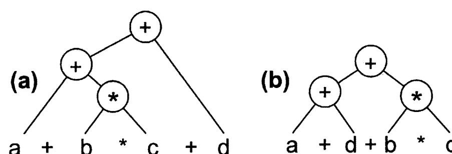

Fig. 7. Performing THR on (a) produces (b).

A. THR

THR was introduced long ago [13], [14] as an optimization method for parallel software compilers. It is a technique to reduce the height of an arithmetic expression tree, where the height of the tree is the number of steps required to compute the expression. In the best case, it achieves the tree height of $O(\log\ n)$ for an expression with n operations. THR uses commutativity, associativity, and distributivity properties of addition, subtraction, and multiplication. In the classical case, THR is achieved at the expense of adding more resources to obtain maximum parallelism in the expression. In previous work for hardware synthesis, THR has been proven useful in high-level synthesis of data-intensive circuits such as DSP and multimedia applications [15]–[17].

In our work, we use THR as an expression tree manipulation technique. THR will achieve the best execution time when using an unlimited number of two input adders, subtracters, and multipliers. Since we are focusing on libraries that have more complex blocks, THR may or may not result in the optimal execution time. The result is dependent on the library components available. Fig. 7 shows an example of how THR can reduce the CPD. Fig. 7(b) is obtained after applying THR on Fig. 7(a).

B. Factor and Expand

As mentioned previously, traditional THR [13], [14] only uses associativity, commutativity, and distributivity to transform expressions. Since we have access to a symbolic manipulation tool in SymSyn, we can benefit from other transformations as well. One such transformation is common subexpression factorization. Factorization can reduce the number of components used as well as the tree height of a given expression. An example is shown in Fig. 8. Factorization transforms the expression shown in Fig. 8(a) to the expression show in Fig. 8(b). Fig. 8(b) has three less multiplications, one less addition, and shorter tree height compared to Fig. 8(a).

Another useful symbolic manipulation technique is expansion. This manipulation technique changes the polynomial into its sum of products format. Meanwhile, it is capable of straightforward simplification techniques that can save both delay and area. A small example of such simplification is transforming $\mathtt{a}+\mathtt{a}+\mathtt{a}$ to $3*\mathtt{a}$.

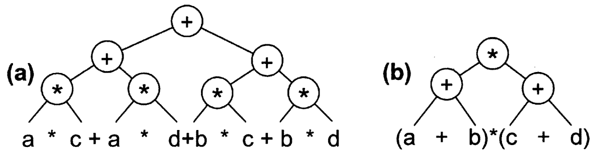

Fig. 8. Factor may reduce the number of components and CPD.

C. Horner Form

The Horner form of a polynomial is a nested normal form with a minimal number of multiplications and additions. Any polynomial can be rewritten in Horner, or nested, form. The general univariate case is defined as follows [8]:

$$\begin{aligned} p(x) &= a_0 \cdot x^n + \cdots + a_{n-1} \cdot x + a_n \\ &= \Big(\cdots \big((a_0 \cdot x + a_1) \cdot x + a_2 \big) \cdot x + \cdots a_{n-1} \Big) \cdot x + a_n. \end{aligned}$$

Assume that x^n can be calculated using only $\log_2(n)$ multiplications for integer n. For a polynomial of degree n, the Horner form requires n multiplications and n additions. The expanded form, however, requires

$$\sum_{i=1}^{n} \log_2(i) = \log_2(n!)$$

multiplications, which is more than twice as expensive for a polynomial of degree 10. Thus, one advantage of the Horner form is that the work involved in exponentiation is distributed across addition and multiplication, resulting in savings of some basic arithmetic operations. Another advantage is that Horner form is more stable to evaluate numerically when compared with the expanded form. This is because each sum or product involves quantities which vary on a more evenly distributed scale [8]. For hardware implementation, the Horner form has a distinct advantage. It effectively maps a univariate polynomial to cost effective multiplier-accumulators (MAC).

Horner form is generalized for multivariate polynomials by specifying an ordered list of variables. As a simple example, consider the following polynomial in which the number of multiplications is reduced from 32 to 13:

```
> S := x^3+3*x^2*y+x^2+3*x*y^2+2*x*y+2*x^2*z+
  y^3 + y^2 + 2*y^2*z + 2*y*z + z^2*x + z^2*y + z^2;
> convert(S,'horner', [x, y, z]);
  z^2 + ((2 + z)*z + (2*z + 1 + y)*y)*y + ((2 + z)*z
  + (2 + 4*z + 3*y)*y + (2*z + 1 + 3*y + x)*x)*x
```

D. Substitution and Elimination

Substitution is defined as replacing a subexpression by a previously computed variable [1]. It reduces complexity of a function by using an additional variable that was not previously in its support set. This transformation creates a new dependency between expressions, but may also eliminate previous dependencies. Substitution has been previously used in multilevel combinational logic optimization [18], [19]. Elimination theory [12] based on the Gröbner basis formalizes substitution and variable elimination for multivariate polynomials. We refer the interested reader to [12] for the detailed mathematical proof. Note that for arithmetic polynomials, use of a more general decomposition model is necessary as compared to the algebraic division modeled in combinational logic synthesis. This is due to the fact that the Boolean impotence property does not hold in arithmetic polynomials and arithmetic polynomials can have exponents. Therefore, there is no restriction on the support set of the divisor and quotient of an expression. For example, $(x^2 - y^2)/(x - y) = x + y$ is a legitimate division.

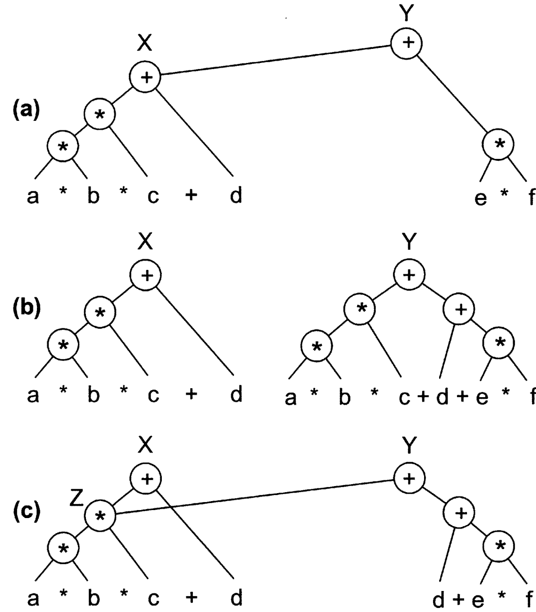

Fig. 9. Substitution with THR can maximize parallelism.

TABLE I
NORMALIZED DELAY AND AREA OF LIBRARY ELEMENTS

Library Element	Delay	Normalized Delay	Area	Normalized Area
Add	7.54	1	15090	1
Square	7.89	1.05	89814	5.95
Mult	10.17	1.35	133401	8.84
MAC	17.28	2.29	142554	9.45
Sine	45.21	6.00	625218	41.43
Cosine	45.37	6.02	622849	41.28
SQRT	21.42	2.84	36031	2.39

Substitution can be combined with THR in order to select a subexpression that maximizes parallelism. As a simple example, let us consider a basic block which consists of two arithmetic expressions

```
X := a*b*c + d
Y := X + e*f.
```

It can be seen that Y is dependent on X, therefore, Y is calculated after the value of X is known as shown in Fig. 9(a). However, if we eliminate X in Y, Y := a * b * c + d + e * f, Y can be evaluated in parallel with X. Fig. 9(b) shows the results of THR on both X and Y expressions. In order to achieve maximum parallelism between X and Y, we now substitute only subexpression

TABLE II
SYMSYN RESULTS FOR SOME EXAMPLES

Data-flow Examples	Lexicographical Mapping			Minimal Component Mapping			Minimal CPD Mapping		
	Num. of components	Area	CPD	Num. of components	Area	CPD	Num. of components	Area	CPD
a^2-b^2	3	18.68	2.35	3	10.84	2.35	3	12.90	2.05
b^3+ba^2c	6	45.20	3.70	4	30.19	4.69	6	39.42	3.70
1-$x0^2$/2+$x0^4$/24+x0+x1x2	11	65.88	5.70	3	51.72	7.02	6	41.24	5.58
cos(sin(x0))	24	180.81	7.40	2	82.71	12.01	9	64.88	6.43
IDCT	9	63.88	4.70	2	15.40	3.34	2	15.40	3.34
anti-alias	27	191.65	9.09	8	60.14	14.55	12	92.61	7.04
Geometric-transform	12	82.56	10.09	2	50.27	8.29	5	43.13	7.92
1/2tanh(a-1)+ 1/2tanh(a+1)	16	94.40	9.74	3	24.85	5.63	4	30.80	4.38
PSK	33	229.01	7.40	2	42.28	7.02	2	42.28	7.02
Turbo decoder	104	817.47	16.14	4	125.30	12.99	4	125.30	12.99
Gabor-transform	79	565.10	12.44	6	96.61	9.41	6	96.61	9.41

$a * b * c$ in Y with a new variable $z := a * b * c$. The result is shown in Fig. 9(c).

VII. IMPLEMENTATION AND EXPERIMENTAL RESULTS

SymSyn is an environment that, used in conjunction with classical high-level synthesis algorithms, can automate efficient synthesis of data flow intensive circuits. It takes as input a data path of the circuit under design and automatically maps that data path to complex library elements, without need of any directives from the designer. The program inputs are polynomial representations of data flow and a set of polynomials representing the library elements. Output is a report of components used to implement the data flow and the way they are connected, such that the CPD or number of components used, is minimized. SymSyn contains implementations of the algorithms described in Sections IV and V and the heuristics described in Section VI as accelerators. The implementation is mainly in C programming language, with calls to Maple V [7] for the symbolic manipulations.

We have tested the efficiency of SymSyn with a number of data-path examples. In our tests, the area and CPD reported are normalized by the area and CPD of a full adder. For example, the CPD of an adder is 1 and CPD of a multiplier is 1.35. This number is calculated from the CPD reported by Synopsys Design Compiler (DC) for a 16-bit multiplier divided by the CPD reported by Synopsys DC for a 16-bit adder. The normalized CPD calculation is done for all library components available in the Synopsys DesignWare arithmetic component library [2]. Normalized area and CPD of several library elements are shown in Table I.

Experimental results are shown in Table II. The first four data flows in Table II are simple benchmark polynomials. The fifth data flow polynomial is a basic block of a one-dimensional inverse discrete cosine transform (IDCT). The next data flow example is the antialias block described in the introduction. IDCT and antialias are widely used in audio and video compression standards such as JPEG, MPEG, and MP3. The geometric transformation is used in graphics for image rotation. The next three examples come from the field of digital communication. One is a bandpass filter in frequency domain. The other performs phase shift keying (PSK) modulation and the third performs turbo decoding. The last example is a data flow segment of the Gabor transform used in neural systems.

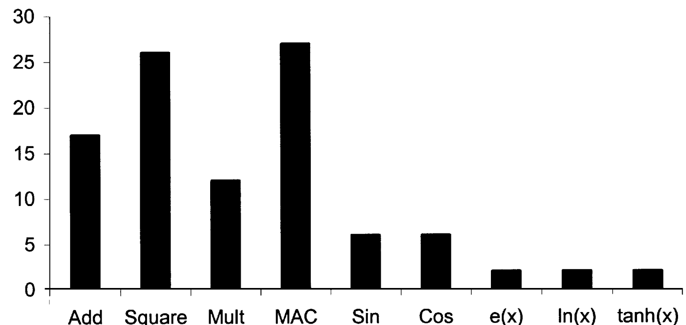

Fig. 10. Component distribution in SymSyn output.

In the first set of results of Table II (lexicographical mapping), we assume that the polynomial representation is mapped only to multipliers and adders. This is the same as the lexicographical component inference that is typical in commercial behavioral synthesis tools. The number of components column shows the numbers of adds and multiplies in the data-path polynomial. The area reported is the area of an adder multiplied by the number of adds, plus the area of a multiplier multiplied by the number of multiplies in the data-path polynomial. The CPD reported is the cumulative delay of components on the critical path.

Next, we map and synthesize the example data flows using SymSyn. The second set of results shown in Table II (*minimal component mapping*), are the results obtained from SymSyn by applying Algorithm 4.1. The mapping reported is the minimal component mapping. We have shown the number of library components Algorithm 4.1 has used in mapping each data

TABLE III
AREA AND DELAY REPORTED BY SYNOPSYS TOOLS USING TSMC.35 LIBRARY

	Synopsys BC results				Synopsys DC results			
	Lexicographical		SymSyn Mapping		Lexicographical		SymSyn Mapping	
Data-flow Examples	Area	Estimated Delay	Area	Estimated Delay	Area	Delay	Area	Delay
A^2-b^2	120295	11	83087	11	66760	11.21	54815	9.42
B^3+ba^2c	469030	24	862816	24	285926	29.09	166303	25.44
1-$x0^2$/2+$x0^4$/24+x0+x1x2	395252	23	137139	16	146526	19.68	93538	14.39
cos(sin(x0))	790784	39	163349	35	314776	38.17	140256	32.47
IDCT	456704	24	178177	18	323185	29.29	130753	20.52
anti-alias	3387672	63	288373	48	1761169	69.43	102357	59.89
Geometric-transform	3051440	39	273340	30	1178868	54.07	190937	25.63
PSK	1812833	36	82705	24	1099991	33.33	80670	21.69

flow polynomial to the extended Synopsys DesignWare library (DesignWare library [2] plus $\tanh(x)$, $\ln(x)$, and $\exp(x)$ operations). Area is the sum of areas of the components used by SymSyn in each data-path implementation.

Finally, the last set of results in Table II (*minimal CPD mapping*), are derived by SymSyn using Algorithm 5.1. The emphasis is to decompose each data flow into the given library such that the CPD of the implementation is minimized. We have reported the number of components and the area and CPD of the implementation suggested by Algorithm 5.1. Note that Algorithm 5.1 maps for maximal parallelism and resource sharing is not used. The CPD reported is sum of the delays of components used in the data-path implementation in view of their data dependencies. Both mapping results shown are using the same component library.

In order to qualify the examples used in Table II, we have shown the distribution of components used in SymSyn output in Fig. 10. Note that the components used most are the MAC operator and the square operator; this result is typical in data-intensive circuits.

In order to obtain a more precise measurement of the CPD and area of our set of examples, we used Synopsys Behavioral Compiler and Synopsys Design Compiler to produce the set of results shown in Table III. The examples in Table III are the subset of examples shown in Table II that did not need $\tanh(x)$, $\ln(x)$, and $\exp(x)$ operations. These operations are not available in the DesignWare library [2]. The *lexicographical* columns correspond to results reported by Synopsys Behavioral Compiler and Design Compiler without any mapping directives in the behavioral HDL code. The *SymSyn mapping* columns are the results reported for the same set of examples when mapping directives suggested by SymSyn are incorporated in the behavioral HDL code. It can be observed that actual performance and area improvements for these examples are inline and better than estimated by SymSyn in Table II.

In summary, the results show that we can achieve an average performance improvement of 25% and an average area improvement of 60% over commercial behavioral synthesis flow. These improvements are the results of intelligent mapping algorithms implemented in SymSyn as opposed to the lexicographical mapping currently available in the commercial tools.

VIII. SUMMARY

This paper has introduced two new decomposition algorithms to map data flow to a set of complex arithmetic library components. These algorithms fit seamlessly in the high-level synthesis flow and enhance the quality of result of data intensive circuit synthesis. Our method takes advantage of two previously developed concepts; one is the polynomial representation of library blocks and the second is symbolic computer algebra. Polynomial representation is used to represent the functionality of library components and the data flow segment of the chip under design. Symbolic computer algebra is used to decompose the data flow to a set of library components. From a practical standpoint, the contribution of this paper is to make arithmetic library binding an automated process, and eliminate the need for user-specified synthesis directives.

Symbolic computer algebra is a powerful set of algorithms not previously used in the field of synthesis. We believe these algorithms open a new set of opportunities in high-level synthesis research. Even though algebraic manipulations are best suited for combinational arithmetic designs, classical scheduling, resource sharing, and retiming algorithms can be applied to the data-path output to achieve optimized/pipelined designs.

The research presented here is especially promising in the fields of graphics, multimedia, and digital signal processing where there is a tolerance for computational error as long as the degradation in audio or video is limited [20]–[22]. This tolerance can be used to approximate nonpolynomials data flows to polynomial representations, which are well-suited inputs for our tool SymSyn. This paper does not explain the approximation tools and truncation errors since there is a wide body of mathematical literature available on these topics [6].

ACKNOWLEDGMENT

The authors would like to thank ARPA/MARCO Gigascale Research Center and Synopsys Inc. for their support.

REFERENCES

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw–Hill, 1994.
[2] DesignWare Library (1994). [Online]. Available: http://www.synopsys.com/

[3] J. Smith and G. De Micheli, “Polynomial methods for component matching and verification,” in *Proc. Int. Conf. Computer-Aided Design*, 1998.

[4] ——, “Polynomial methods for allocating complex components,” in *Proc. Design, Automation, Test Eur. Conf.*, 1999.

[5] ——, “Polynomial circuit models for component matching in high-level synthesis,” *IEEE Trans. VLSI*, vol. 9, pp. 783–800, Dec. 2001.

[6] J. F. Hart *et al.*, *Computer Approximations*. New York: Wiley, 1968.

[7] Maple (1988). [Online]. Available: http://www.maplesoft.com

[8] Mathematica (1987). [Online]. Available: http://www.wri.com

[9] B. Buchberger, “Some properties of Gröbner bases for polynomial ideals,” *ACM SIG-SAM Bullet.*, 1976.

[10] K. Geddes, S. Czapor, and G. Labahn, *Algorithms for Computer Algebra*. Norwell, MA: Kluwer, 1992.

[11] T. Becker and V. Weispfenning, *Gröbner Bases*. New York: Springer-Verlag, 1993.

[12] D. Cox, J. Little, and D. O’Shea, *Ideals, Varieties, and Algorithms*. New York: Springer-Verlag, 1997.

[13] D. J. Kuck, *The Structure of Computers and Computations*. New York: Wiley, 1978, vol. I.

[14] D. J. Kuck, Y. Muraoka, and S. C. Chen, “On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup,” *IEEE Trans. Comput.*, vol. C-21, Dec. 1972.

[15] A. Nicolau and R. Potasman, “Incremental tree height reduction for high level synthesis,” in *Proc. Design Automation Conf.*, 1991, pp. 770–774.

[16] D. Kolson, A. Nicolau, and N. Dutt, “Integrating program transformations in the memory-based synthesis of image and video algorithms,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994.

[17] H. Wang, A. Nicolau, and K. Siu, “The strict time lower bound and optimal schedules for parallel prefix with resource constraints,” *IEEE Trans. Comput.*, Nov. 1996.

[18] R. Brayton and C. McMullen, “The decomposition and factorization of logic synthesis,” in *Proc. IEEE Int. Symp. Circuits Syst.*, May 1982.

[19] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, “MIS: A multiple-level logic optimization and the rectangular covering problem,” in *Proc. Int. Conf. Computer-Aided Design*, 1987.

[20] M. Willems, H. Keding, T. Grötket, and H. Meyr, “Fridge: An interactive fixed-point code generation environment for HW/SW CoDesign,” in *Proc. Int. Conf. Acoustics, Speech, Signal Process.*, 1997.

[21] G. Constantinides, P. Cheung, and W. Luk, “Heuristic datapath allocation for multiple wordlength systems,” in *Proc. Design, Automation Test Eur.*, 2001.

[22] D. Menard, D. Chillet, F. Charot, and O. Sentieys, “Automatic floating-point to fixed-point conversion for DSP code generation,” in *Proc. Int. Conf. Compilers, Architecture, Synthesis Embedded Syst.*, 2002.

Armita Peymandoust received the B.S. degree in electrical and computer engineering from University of Tehran, Tehran, Iran, the M.S. degree in electrical and computer engineering from Northeastern University, Boston, MA, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, 2003.

Previously, she was a Design Engineer on the IA-64 product line with the Intel Corporation, Santa Clara, CA. She is currently a Senior Research and Development Engineer with Synopsys, Inc., Mountain View, CA. Her research interests include embedded systems, system-level design and synthesis, hardware/software codesign, and computer architecture.

Giovanni De Micheli (S’79–M’79–SM’80–F’94) received the nuclear engineer degree from Politecnico di Milano, Milan, Italy, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is a Professor of electrical engineering and, by courtesy, of computer science at Stanford University, Stanford, CA. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software co-design and low-power design. He is author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw–Hill, 1994) and co-author and/or co-editor of five other books and of over 250 technical articles.

Dr. De Micheli is a Fellow of ACM. He received the 2003 IEEE Emanuel Priore Award for contributions to computer-aided synthesis of digital systems. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and 1993. He is President of the IEEE CAS Society. He was Editor in Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS from 1987 to 2001. He was the Program and General Chair of the Design Automation Conference (DAC) from 1996 to 1997 and 2000, respectively. He was the Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was a founding member of the ALaRI institute at Universita’ della Svizzera Italiana (USI), in Lugano, Switzerland, where he is currently scientific counselor.