Chapter 14
Algebraic Techniques to Enhance Common Sub-expression Extraction for Polynomial System Synthesis

Sivaram Gopalakrishnan and Priyank Kalla

Abstract Datapath designs that perform polynomial computations over bit-vectors are found in many practical applications, such as in Digital Signal Processing, communication, multi-media, and other embedded systems. With the growing market for such applications, advancements in synthesis and optimization techniques for polynomial datapaths are desirable. Common sub-expression extraction (CSE) serves as a useful optimization technique in the synthesis of such polynomial systems. However, CSE has limited potential for optimization when many common sub-expressions are not exposed in the given symbolic representation. Given a suitable set of transformations (or decompositions) that expose many common sub-expressions, subsequent application of CSE can offer a higher degree of optimization. This chapter proposes algebraic (algorithmic) techniques to perform such transformations and presents a methodology for their integration with CSE. Experimental results show that designs synthesized using our integrated approach are significantly more area-efficient than those synthesized using contemporary techniques.

14.1 Introduction

High-level descriptions of arithmetic datapaths that perform *polynomial computations* over bit-vectors are found in many practical applications, such as in Digital Signal Processing (DSP) for multi-media applications and embedded systems. These polynomial designs are initially specified using behavioral or Register-Transfer-Level (RTL) descriptions, which are subsequently synthesized into hardware using high-level and logic synthesis tools [23]. With the widespread use of

S. Gopalakrishnan (✉)
Synopsys Inc., Hillsboro, Oregon, USA
e-mail: sivaram.gopalakrishnan@synopsys.com

Based on Gopalakrishnan, S.; Kalla, P.; "Algebraic techniques to enhance common sub-expression elimination for polynomial system synthesis," Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09, pp.1452–1457, 20–24 April 2009 © [2009] IEEE.

S.P. Khatri, K. Gulati (eds.), *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, DOI 10.1007/978-1-4419-7518-8_14,
© Springer Science+Business Media, LLC 2011

such designs, there is a growing need to develop more sophisticated synthesis and optimization techniques for polynomial datapaths at high-level/RTL.

The general area of high-level synthesis has seen extensive research over the years. Various algorithmic techniques have been devised, and CAD tools have been developed that are quite adept at capturing hardware description language (HDL) models and mapping them into control/data-flow graphs (CDFGs), performing scheduling, resource allocation and sharing, binding, retiming, etc. [7]. However, these tools lack the mathematical wherewithal to perform sophisticated algebraic manipulation for arithmetic datapath-intensive designs. Such designs implement a sequence of ADD, MULT type of algebraic computations over bit-vectors; they are generally modeled at RTL or behavioral-level as *systems of multivariate polynomials of finite degree* [19, 22]. Hence, there has been increasing interest in exploring the use of algebraic manipulation of polynomial expressions, for RTL synthesis of arithmetic datapaths. Several techniques such as Horner decomposition, factoring with common sub-expression extraction [13], term-rewriting [1] have been proposed. Symbolic computer algebra [10, 19, 22] has also been employed for polynomial datapath optimization. While these methods are useful as stand-alone techniques, they exhibit limited potential for optimization as explained below.

Typically, in a system of polynomials representing an arithmetic datapath, there are many common sub-expressions. In such systems, common sub-expression extraction (CSE) serves as a useful optimization technique, where isomorphic patterns in an arithmetic expression tree are identified, extracted, and merged. This prevents the cost of implementing multiple copies of the same expression. However, CSE has a limited potential for optimization if the common expressions are not exposed in the given symbolic representation. Hence, application of a "suitable set of transformations" (or decompositions) of the given polynomial representation to expose more common sub-expressions offers a higher potential for optimization by CSE. The objective of this chapter is to develop *algorithmic and algebraic* techniques to perform such transformations, to present a methodology for their integration with CSE, and to achieve a higher degree of optimization.

14.1.1 Motivation

Consider the various decompositions for a system of polynomials P_1, P_2, and P_3, implemented with variables x, y, and z, as shown in Table 14.1. The direct implementation of this system will require 17 multipliers and 4 adders. To reduce the size of the implementation, a Horner-form decomposition may be used. This implementation requires the use of 15 multipliers and 4 adders. However, a more sophisticated factoring method employing kernel/co-kernel extraction with CSE [13, 14] can further reduce the size of the implementation, using 12 multipliers and 4 adders. Now, consider the proposed decomposition of the system, also shown in the table. This implementation requires only 8 multipliers and 1 adder. Clearly, this is an efficient implementation of the polynomial system. This decomposition achieves a high degree of optimization by analyzing common sub-expressions across multiple

Table 14.1 Various decompositions for a polynomial system

Original system	Horner-form decomposition
$P_1 = x^2 + 6xy + 9y^2$; $P_2 = 4xy^2 + 12y^3$; $P_3 = 2x^2z + 6xyz$;	$P_1 = x(x + 6y) + 9y^2$; $P_2 = 4xy^2 + 12y^3$; $P_3 = x(2xz + 6yz)$;
Factorization + CSE	Proposed decompsoition
$P_1 = x(x + 6y) + 9y^2$; $P_2 = y^2(4x + 12y)$; $P_3 = xz(2x + 6y)$;	$d_1 = x + 3y$; $P_1 = {d_1}^2$; $P_2 = 4y^2 d_1$; $P_3 = 2xzd_1$;

polynomials. This is not a trivial task and is not achieved by any earlier manipulation techniques [13, 14]. Note that d_1 is a good building block (common sub-expression) for these system of equations. Identifying and factoring out such building blocks across multiple polynomial datapaths can yield area-efficient hardware implementations.

14.1.2 Contributions

In this chapter, we develop techniques to transform the given system of polynomials by employing certain algebraic manipulations. These transformations have the potential to expose more common terms among the polynomials. These terms can be easily identified by the CSE routines and can be used as good "building blocks" for the design. Our expression manipulations are based on the following algebraic concepts:

- Canonical representation of polynomial functions over finite integer rings of the type Z_{2^m} [4]
- Square-free factorization
- Common coefficient extraction
- Factoring with kernel/co-kernel computation
- Algebraic division

We show how the above-mentioned algebraic methods are developed and employed in a synergistic fashion. These methods form the foundation of an integrated CSE technique for area-efficient implementations of the polynomial system.

14.1.3 Paper Organization

The next section presents the previous work in the area of polynomial datapath synthesis. Section 14.3 describes some preliminary concepts related to polynomial functions and their algebraic manipulations. Section 14.4 describes the optimization methods developed in this chapter. Section 14.5 presents our overall integrated approach. The experimental results are presented in Section 14.6. Finally, Section 14.7 concludes the chapter.

14.2 Previous Work

Contemporary high-level synthesis tools are quite adept in extracting control/data-flow graphs (CDFGs) from the given RTL descriptions and also in performing scheduling, resource-sharing, retiming, and control synthesis. However, they are limited in their capability to employ sophisticated algebraic manipulations to reduce the cost of the implementation. For this reason, there has been increasing interest in exploring the use of algebraic methods for RTL synthesis of arithmetic datapaths.

In [20, 21], the authors derive new polynomial models of complex computational blocks by the way of polynomial approximation for efficient synthesis. In [19], symbolic computer algebra tools are used to search for a decomposition of a given polynomial according to available components in a design library, using a Buchberger-variant algorithm [2, 3] for Gröbner bases. Other algebraic transforms have also been explored for efficient hardware synthesis: factoring with common sub-expression elimination [13], exploiting the structure of arithmetic circuits [24], term re-writing [1], etc. Similar algebraic transforms are also applied in the area of code optimization. These include reducing the height of the operator trees [18], loop expansion, induction variable elimination. A good review of these approaches can be found in [8].

Taylor Expansion Diagrams (TEDs) [5] have also been used for data-flow transformations in [9]. In this technique, the arithmetic expression is represented as a TED. Given an objective (design constraint), a sequence of decomposition cuts are applied to the TED that transforms it to an optimized data-flow graph. Modulo arithmetic has also been applied for polynomial optimization/decomposition of arithmetic datapaths in [10, 11]. By accounting for the bit-vector size of the computation, the systems are modeled as polynomial functions over finite integer rings. Datapath optimization is subsequently performed by exploiting the number theoretic properties of such rings, along with computational commutative algebra concepts.

14.2.1 Kernel/Co-kernel Extraction

Polynomial systems can be manipulated by extracting common expressions by using the kernel/co-kernel factoring. The work of [13] integrates factoring using kernel/co-kernel extraction with CSE. However, this approach has its limitations. Let us understand the general methodology of this approach before describing its limitations. The following terminologies are mostly referred from [13].

A **literal** is a variable or a constant. A **cube** is a product of variables raised to a non-negative integer power, with an associated sign. For example, $+acb$, $-5cde$, $-7a^2bd^3$ are cubes. A sum of product (SOP) is said to be cube-free if no cube (except “1”) divides all the cubes of the SOP. For a polynomial P and a cube c, the expression P/c is a **kernel** if it is cube-free and has at least two terms. For example, when $P = 4abc - 3a^2b^2c$, the expression $P/abc = 4 - 3ab$ is a **kernel**. The cube that is used to obtain the kernel is the **co-kernel** (abc). This approach has two major limitations:

Coefficient Factoring: Numeric coefficients are treated as literals, not numbers. For example, consider a polynomial $P = 5x^2 + 10y^3 + 15pq$. According to this approach, coefficients $\{5, 10, 15\}$ are also treated as literals like variables $\{x, y, p, q\}$. Since it does not use algebraic division, it cannot determine the following decomposition: $P = 5(x^2 + 2y^3 + 3pq)$.

Symbolic Methods: Polynomials are factored without regard to their algebraic properties. Consider a polynomial $P = x^2 + 2xy + y^2$, which can actually be transformed as $(x + y)^2$. Such a decomposition is also not identified by this kernel/co-kernel factoring approach. The reason for the inability to perform such a decomposition is due to the lack of symbolic computer algebra manipulation.

This chapter develops certain algebraic techniques that address these limitations. These techniques, along with kernel/co-kernel factoring, can be seamlessly integrated with CSE to provide an additional degree of optimization. With this integration, we seek to extend the optimization potential offered by the conventional methods.

14.3 Preliminary Concepts

This section will review some fundamental concepts of factorization and polynomial function manipulation, mostly referred from [4, 6].

14.3.1 Polynomial Functions and Their Canonical Representations

A bit-vector of size m represents integer values reduced modulo 2^m. Therefore, polynomial datapaths can be considered as polynomial functions over finite integer rings of the form Z_{2^m}. Moreover, polynomial datapaths often implement bit-vector arithmetic with operands of different bit-widths. Let $x_1, \ldots, x_d$ represent the bit-vector variables, where each bit-vector has bit-width $n_1, \ldots, n_d$. Let f be the bit-vector output of the datapath, with m as its bit-width. Then the bit-vector polynomial can be considered as a function $f : Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}} \rightarrow Z_{2^m}$.

A function f from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}} \rightarrow Z_{2^m}$ is said to be a polynomial function if it is represented by a polynomial $F \in Z[x_1, x_2, \ldots, x_d]$; i.e., $f(x_1, x_2, \ldots, x_d) \equiv F(x_1, x_2, \ldots, x_d)$ for all $x_i \in Z_{2^{n_i}}$, $i = 1, 2, \ldots, d$ and $\equiv$ denotes congruence (mod 2^m).

Let $f : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ be a function defined as: $f(0, 0) = 1$, $f(0, 1) = 3$, $f(0, 2) = 5$, $f(0, 3) = 7$, $f(1, 0) = 1$, $f(1, 1) = 4$, $f(1, 2) = 1$, $f(1, 3) = 0$. Then, f is a polynomial function representable by $F = 1 + 2y + xy^2$, since $f(x, y) \equiv F(x, y) \bmod 2^3$ for $x = 0, 1$ and $y = 0, 1, 2, 3$.

Polynomial functions implemented over specific bit-vector sizes can be represented in a unique canonical form. According to [4, 10], any polynomial representation F for a function f, from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots Z_{2^{n_d}}$ to Z_{2^m}, can be uniquely represented as a sum-of-product of falling factorial terms:

$$F = \Sigma_{\mathbf{k}} c_{\mathbf{k}} \mathbf{Y}_{\mathbf{k}} \tag{14.1}$$

where,

- $\mathbf{k} =< k_1, \ldots, k_d >$ for each $k_i = 0, 1, \ldots, \mu_i - 1$;
- $\mu_i = \min(2^{n_i}, \lambda)$, for each $i = 1, \ldots, d$;
- λ is the least integer such that 2^m divides $\lambda!$;
- $c_{\mathbf{k}} \in Z$ such that $0 \leq c_{\mathbf{k}} < \frac{2^m}{gcd(2^m, \prod_{i=1}^{d} k_i!)}$;

In (14.1), $\mathbf{Y_k}$ is represented as

$$\mathbf{Y_k}(\mathbf{x}) = \prod_{i=1}^{d} Y_{k_i}(x_i)$$
$$= Y_{k_1}(x_1) \cdot Y_{k_2}(x_2) \cdots Y_{k_d}(x_d) \quad (14.2)$$

where $Y_k(x)$ is a falling factorial defined as follows:

Definition 14.1 Falling factorials of degree k are defined according to:

- $Y_0(x) = 1$
- $Y_1(x) = x$
- $Y_2(x) = x(x-1)$

  $\vdots$
- $Y_k(x) = (x - k + 1) \cdot Y_{k-1}(x)$

Intuitively, this suggests that while having a canonical form representation as in (14.1), it is possible to find common $Y_{k_i}(x_i)$ terms.

For example, consider the following polynomials implementing a 16-bit datapath, i.e., as polynomial functions over $f : Z_{2^{16}} \times Z_{2^{16}} \rightarrow Z_{2^{16}}$:

$$F = 4x^2y^2 - 4x^2y - 4xy^2 + 4xy + 5z^2x - 5zx \quad (14.3)$$
$$G = 7x^2z^2 - 7x^2z - 7xz^2 + 7zx + 3y^2x - 3yx \quad (14.4)$$

Using the canonical form representation, we get

$$F = 4Y_2(x)Y_2(y) + 5Y_2(z)Y_1(x) \quad (14.5)$$
$$G = 7Y_2(x)Y_2(z) + 3Y_2(y)Y_1(x) \quad (14.6)$$

Such a representation exposes many common terms in $Y_{k_i}(x_i)$. These terms may subsequently serve as a good basis for common sub-expression extraction.

For a detailed description of the above canonical form representation, the canonical reduction operations, and their impact on hardware implementation costs for polynomial datapaths, the reader is referred to [10].

14.3.2 Factorization

Definition 14.2 Square-free polynomial Let F be a field or an integral domain Z. A polynomial u in $F[x]$ is a square-free polynomial if there is no polynomial v in $F[x]$ with $\deg(v, x) > 0$, such that $v^2|u$.

Although the definition is expressed in terms of a squared factor, it implies that the polynomial does not have a factor of the form v^n with $n \geq 2$.

Example 14.1 The polynomial $u_1 = x^2 + 3x + 2 = (x + 1)(x + 2)$ is square-free. However, $u_2 = x^4 + 7x^3 + 18x^2 + 20x + 8 = (x + 1)(x + 2)^2$ is not square-free, as v^2 (where $v = x + 2$) divides u_2.

Definition 14.3 Square-free factorization A polynomial u in $F[x]$ has a unique factorization

$$u = c s_1 {s_2}^2 \cdots {s_m}^m \tag{14.7}$$

where c is in F and each s_i is monic and square-free with $gcd(s_i, s_j) = 1$ for $i \neq j$. This unique factorization in (14.7) is called square-free factorization of u.

Example 14.2 The polynomial $u = 2x^7 - 2x^6 + 24x^5 - 24x^4 + 96x^3 - 96x^2 + 128x - 128$ has a square-free factorization $2(x - 1)(x^2 + 4)^3$ where $c = 2$, $s_1 = x - 1$, $s_2 = 1$, and $s_3 = x^2 + 4$. Note that a square-free factorization may not contain all the powers given in (14.7).

A square-free factorization only involves the square-free factors of a polynomial and leaves the deeper structure that involves the irreducible factors intact.

Example 14.3 Using square-free factorization

$$x^6 - 9x^4 + 24x^2 - 16 = (x^2 - 1)(x^2 - 4)^2 \tag{14.8}$$

both factors are reducible. This suggests that even after obtaining square-free polynomials, there is a potential for additional factorization. In other words, consider 14.8, where $(x^2 - 1)$ can be further factored as $(x + 1)(x - 1)$ and $(x^2 - 4)^2$ can be factored as $((x + 2)(x - 2))^2$.

14.4 Optimization Methods

The limitations of contemporary techniques come from their narrow approach to factorization, relying on single types of factorization, instead of the myriad of optimization techniques available. We propose an integrated approach, to polynomial optimization, to overcome these limitations. This section describes the various optimization techniques that are developed/employed in this chapter.

14.4.1 Common Coefficient Extraction

The presence of many coefficient multiplications in polynomial systems increases the area-cost of the hardware implementation. Moreover, existing coefficient factoring techniques [13] are inefficient in their algebraic manipulation capabilities. Therefore, it is our focus to develop a coefficient factoring technique that employs efficient algebraic manipulations and as a result reduces the number of coefficient multiplications in the given system.

Consider the following polynomial $P_1 = 8x + 16y + 24z$. When coefficient extraction is performed over P_1, it results in three possible transformations, given as follows:

$$P_1 = 2(4x + 8y + 12z) \tag{14.9}$$

$$P_1 = 4(2x + 4y + 6z) \tag{14.10}$$

$$P_1 = 8(x + 2y + 3z) \tag{14.11}$$

From these three transformations, (14.11) extracts the highest common term in P_1. This results in the best transformation (reduced set of operations). A method to determine the highest common coefficient is the greatest common divisor (GCD) computation. Therefore, in this approach, GCD computations are employed to perform common coefficient extraction (CCE) for a system of polynomials. The pseudocode to perform CCE is shown in Algorithm 6.

Algorithm 6 Common Coefficient Extraction (CCE)

```
1:  CCE(a_1, ···, a_n)
2:  /* (a_1, ···, a_n) = Coefficients of the given polynomial;*/
3:  for every pair (a_i, a_j) in n do
4:      Compute GCD(a_i, a_j);
5:      Ignore GCDs = "1";
6:      if GCD(a_i, a_j) < a_i and GCD(a_i, a_j) < a_j then
7:          Ignore the GCDs;
8:      end if
9:  end for
10: Order the GCDs in decreasing order;
11: while GCD list is non-empty do
12:     Perform the extraction using that order
13:     Store the linear/non-linear blocks created as a result of extraction
14:     Remove GCDs corresponding to extracted terms and update the GCD list
15: end while
```

Let us illustrate the operation of the CCE routine. Consider the polynomial P_1 computed as

$$P_1 = 8x + 16y + 24z + 15a + 30b + 11 \tag{14.12}$$

The input to CCE is the coefficients of the given polynomial that are involved in coefficient multiplications. In other words, if there is a coefficient addition in the polynomial, it is not considered while performing CCE. For example in (14.12), only the coefficients {8, 16, 24, 15, 30} are considered and 11 is ignored. The reason is because there is no benefit in extracting this coefficient and a direct implementation is the cheapest in terms of area-cost.

The algorithm then begins by computing the GCDs for every pair-wise combination of the coefficients in the input set. Computing pair-wise GCDs of the coefficients:

$$\begin{aligned} GCD(8, 16) &= 8 \\ GCD(8, 24) &= 8 \\ &\vdots \\ GCD(15, 30) &= 15 \end{aligned} \tag{14.13}$$

we get the following set {8, 8, 1, 2, 8, 1, 2, 3, 6, 15}. However, only a subset is generated by ignoring "GCDs = 1" and "GCDs $(a_i, a_j) < a_i$ and a_j." The reason for ignoring these GCDs is that we only want to extract the highest common coefficients that would result in a reduced cost. For example, the GCD(24, 30) = 6. However, extracting 6 does not reduce the cost of the sub-expression $24z + 30b$ in (14.12), as $6(4z + 5b)$ requires more coefficient multipliers.

Applying the above concepts, the final subset is {8, 15}. This set is then arranged in the decreasing order to get {15, 8}. The first element is "15." On performing the extraction using coefficient "15," the following decomposition is realized:

$$P_1 = 8x + 16y + 24z + 15(a + 2b) \tag{14.14}$$

This creates a smaller polynomial $(a + 2b)$. It should be noted that this is a linear polynomial. This polynomial is stored and the extraction continues until the GCD list is empty. After CCE, the polynomial decomposition obtained is

$$P_1 = 8(x + 2y + 3z) + 15(a + 2b) \tag{14.15}$$

Two linear blocks $(a + 2b)$ and $(x + 2y + 3z)$ are finally obtained. The motivation behind storing these polynomials is that they can serve as potentially good building blocks in the subsequent optimization methods.

14.4.2 Common Cube Extraction

Common cubes, that consist of products of variables, also need to be extracted from the given polynomial representation. The kernel/co-kernel extraction technique from [13] is quite efficient for this purpose. Therefore, we employ this approach

to perform the common cube extraction. Note that the cube extraction technique of [13] also considers coefficients as variables. We do not allow the technique of [13] to treat coefficients as variables – as we employ CCE for coefficient extraction. We employ this technique of [13] for extracting cubes composed only of variables.

Consider the following system of polynomials:

$$\begin{aligned} P_1 &= x^2y + xyz \\ P_2 &= ab^2c^3 + b^2c^2x \\ P_3 &= axz + x^2z^2b \end{aligned} \tag{14.16}$$

A kernel/co-kernel cube extraction results in the following representation. (Here, c_k is the co-kernel cube and k is the kernel.)

$$\begin{aligned} P_1 &= (xy)_{ck}(x + z)_k \\ P_2 &= (b^2c^2)_{ck}(ac + x)_k \\ P_3 &= (xz)_{ck}(a + xzb)_k \end{aligned} \tag{14.17}$$

Note that this procedure (which we call Cube_Ex()) exposes both cubes and kernels as potential (common) building blocks, which CSE can further identify and extract.

14.4.3 Algebraic Division

This method can potentially lead to a high degree of optimization. The problem essentially lies in identifying a good divisor, which can lead to an efficient decomposition. Given a polynomial $a(x)$, and a set of divisors $(b_i(x))$, $\forall i$ we can perform the division $a(x)/b_i(x)$ and determine if the resulting transformation is optimized for hardware implementation.

Using common coefficient extraction and cube extraction , a large number of linear blocks, that are simpler than the original polynomial, are exposed. These linear blocks can subsequently be used for performing algebraic division. For our overall synthesis approach, we consider only the exposed “linear expressions” as algebraic divisors. The motivation behind using the exposed “linear” blocks for division is that

- Linear blocks cannot be decomposed any further, implying that they have to be certainly implemented.
- They also serve as good building blocks in terms of (cheaper) hardware implementation.

For example, using cube extraction the given system in Table 14.1 is transformed to

$$P_1 = x(x+6y) + 9y^2 \quad or \quad P_1 = x^2 + y(6x+9y)$$
$$P_2 = 4y^2(x+3y)$$
$$P_3 = 2xz(x+3y) \tag{14.18}$$

The following linear blocks are now exposed: $\{(x+6y), (6x+9y), (x+3y)\}$. Using these blocks as divisors, we divide P_1, P_2, and P_3. $(x+3y)$ serves as a good building-block because it divides all the three polynomials as

$$P_1 = (x+3y)^2$$
$$P_2 = 4y^2(x+3y)$$
$$P_3 = 2xz(x+3y) \tag{14.19}$$

Such a transformation to (14.19) is possible only through algebraic division. None of the other expression manipulation techniques can identify this transformation.

14.5 Integrated Approach

The overall approach to polynomial system synthesis is presented in this section. We show how we integrate the algebraic methods presented previously with common sub-expression elimination. The pseudocode for the overall integrated approach is presented in Algorithm 7.

The algorithm operates as follows:

- The given system of polynomials is initially stored in a list of arrays. Each element in the list represents a polynomial. The elements in the array for each list represent the transformed representations of the polynomial. Figure 14.1a shows the polynomial data structure representing the system of polynomials in its expanded form, canonical form (can), and square-free factored form (sqf).
- The algorithm begins by computing the canonical forms and the square-free factored forms, for all the polynomials in the given system. At this stage, the polynomial data structure looks like in Fig. 14.1a.
- Then, the best-cost implementation among these representations is chosen and stored as P_{initial}. The cost is stored as C_{initial}. We estimate the cost using the number of adders and multipliers required to implement the polynomial.
- Common coefficient extraction (CCE) and common cube extraction (Cub_Ex) are subsequently performed. The linear/non-linear polynomials obtained from these extractions are stored/updated. Also, the resulting transformations for each polynomial are updated in the polynomial data structure. At this stage, the data structure looks like in Fig. 14.1b. To elaborate further, in this figure, $\{P_1, P_{1a}, P_{1b}, P_{1c}\}$ are various representations of P_1 (as a result of CCE and Cub_Ex), and so on.
- Using the linear blocks, algebraic division is performed and the polynomial data-structure is further populated, with multiple representations.

Algorithm 7 Approach to Polynomial System Synthesis

```
1: /*Given: (P_1, P_2, ..., P_n) = Polys (P_i's) representing the system; Each P_i is a list to store
   multiple representations of P_i;*/
2: Poly_Synth(P_1, P_2, ..., P_n)
3: /*Initial set of Polynomials, P_orig*/
4: P_orig = ⟨P_1, ..., P_n⟩;
5: P_can = Canonize(P_orig);
6: P_sqf = Sqr_free(P_orig);
7: Initial_cost C_initial = min_cost(P_orig, P_can, P_sqf);
8: /*The polynomial with cost C_initial is P_initial*/
9: CCE(P_initial); Update resulting linear/non-linear polynomials;
10: /*P_CCE = Polynomial representation after CCE();*/   Update P_i's;
11: Cube_Ex(P_i's); Update resulting linear/non-linear polynomials;
12: /*P_CCE_Cube = Polynomial representation after Cube_Ex();*/   Update P_i's;
13: Linear polynomials exposed are lin_poly =⟨l_1, ..., l_k⟩
14: for every l_j in lin_poly do
15:     ALG_DIV(P_i's, l_j);
16:     Update P_i's and l_j's;
17: end for
18: for every combination of P_i's (P_comb) representing P_orig do
19:     Cost = CSE(P_comb);
20:     if (Cost < C_initial) then
21:         C_initial = Cost;
22:         P_final = P_comb;
23:     end if
24: end for
25: return P_final;
```

- The entire polynomial system can be represented using a list of polynomials, where each element in the list is some representation for each polynomial. For example, $\{P_1, P_{2a}, P_{3b}\}$ is one possible list that represents the entire system (refer Fig. 14.1b). The various lists that represent the entire system are given by

$$\begin{gathered}\{(P_1, P_2, P_3),\ \ (P_1, P_2, P_{3a}),\ \ (P_1, P_2, P_{3b}),\\ \vdots \\ (P_{1a}, P_{2b}, P_3),\ (P_{1a}, P_{2b}, P_{3a}),\ (P_{1a}, P_{2b}, P_{3b}),\\ \vdots \\ (P_{1c}, P_{2b}, P_3),\ (P_{1c}, P_{2b}, P_{3a}),\ (P_{1c}, P_{2b}, P_{3b})\}\end{gathered} \tag{14.20}$$

- Finally, we can pick the decomposition with the least estimated cost. For example, Fig. 14.1c shows that the least-cost implementation of the system is identified as:

$$P_{final} = (P_{1a}, P_{2b}, P_{3a}) \tag{14.21}$$

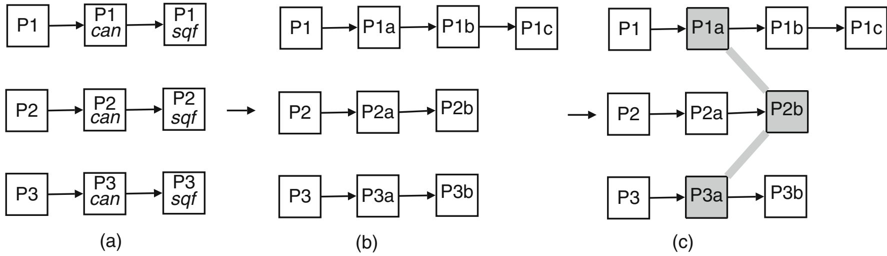

Fig. 14.1 Polynomial system representations

The working of Algorithm 7 is explained with the polynomial system presented in Table 14.2.

Table 14.2 Illustration of algorithm 7

Original system
$P_1 = 13x^2 + 26xy + 13y^2 + 7x - 7y + 11;$ $P_2 = 15x^2 - 30xy + 15y^2 + 11x + 11y + 9;$ $P_3 = 5x^3y^2 - 5x^3y - 15x^2y^2 + 15x^2y + 10xy^2 - 10xy + 3z^2;$ $P_4 = 3x^2y^2 - 3x^2y - 3xy^2 + 3xy + z + 1;$
After canonization and CCE
$P_1 = 13(x^2 + 2xy + y^2) + 7(x - y) + 11;$ $P_2 = 15(x^2 - 2xy + y^2) + 11(x + y) + 9;$ $P_3 = 5x(x - 1)(x - 2)y(y - 1) + 3z^2;$ $P_4 = 3x(x - 1)y(y - 1) + z + 1;$
After cube extraction
$P_1 = 13(x(x + 2y) + y^2) + 7(x - y) + 11;$ $P_2 = 15(x(x - 2y) + y^2) + 11(x + y) + 9;$ $P_3 = 5x(x - 1)(x - 2)y(y - 1) + 3z^2;$ $P_4 = 3x(x - 1)y(y - 1) + z + 1;$
Final decomposition
$d_1 = x + y; d_2 = x - y; d_3 = x(x - 1)y(y - 1)$ $P_1 = 13(d_1{}^2) + 7d_2 + 11;\ P_2 = 15(d_2{}^2) + 11d_1 + 9;$ $P_3 = 5d_3(x - 2) + 3z^2;\ P_4 = 3d_3 + z + 1;$

Initially, canonical reduction and square-free factorization are performed. In this example, this technique does not result in any decomposition for square-free factorization. For P_3 and P_4, there is a low-cost canonical representation.

We then compute the initial cost of the polynomial by using only CSE. In the original system, there are no common sub-expressions. The total cost of the original system is estimated as 51 MULTs and 21 ADDs. Then CCE is performed, resulting in the transformation, as shown in the Table 14.2.

The linear polynomials obtained are $(x - y)$ and $(x + y)$. The non-linear polynomials are $(x^2 + 2xy + y^2)$ and $(x^2 - 2xy + y^2)$. After performing common cube extraction (Cube_Ex()), the additional linear blocks added are $(x+2y)$ and $(x-2y)$.

Subsequently, algebraic division is applied using the linear blocks as divisors for all representations of the polynomial system. The final decomposition with CSE leads to an implementation where only the linear blocks $(x + y)$ and $(x - y)$ are used. The representation for the final implementation is shown in the final row of Table 14.2. The total cost of the final implementation is 14 MULTs and 12 ADDs.

14.6 Experiments

The datapath computations are provided as a polynomial system, operating over specific input/output bit-vector sizes. All algebraic manipulations are implemented in Maple [15]; however, for Horner-form decomposition and factorization, we used the routines available in MATLAB [17]. For common sub-expression elimination, we use the JuanCSE tool available at [14]. Based on the given decomposition (for each polynomial in the system), the individual blocks are generated using the Synopsys Design Compiler [23]. These units are subsequently used to implement the entire system.

The experiments are performed on a variety of DSP benchmarks. The results are presented in Table 14.3. The first column lists the polynomial systems used for the experiments. The first five benchmarks are Savitzky-Golay filters. These filters are widely used in image-processing applications. The next benchmark is a polynomial system implementing quadratic filters from [16]. The next benchmark is from [12], used in automotive applications. The final benchmark is a multi-variate cosine wavelet used in graphics application from [13]. In the second column, we list the design characteristics: number of variables (bit-vectors), the order (highest degree), and the output bit-vector size (m). Column 3 lists the number of polynomials representing the entire system. Columns 4 and 5 list the implementation area and delay, respectively, of the polynomial system implemented using Factorization + common sub-expression elimination. Columns 6 and 7 list the implementation area and delay of the polynomial system, implemented using our proposed method. Columns 8 and 9 list the improvement in the implementation area and delay using our polynomial decomposition technique, respectively. Considering all the benchmarks, we show

Table 14.3 Comparison of proposed method with factorization/CSE

			Factorization/CSE		Proposed method		Improvement	
Systems	Var/Deg/m	# polys	Area	Delay	Area	Delay	Area %	Delay %
SG_3X2	2/2/16	9	204805	186.6	102386	146.8	50	21.3
SG_4X2	2/2/16	16	449063	211.7	197599	262.8	55.9	−24.1
SG_4X3	2/3/16	16	690208	282.3	557252	328.5	19.2	−16.3
SG_5X2	2/2/16	25	570384	205.6	271729	234.2	52.3	−13.9
SG_5X3	2/3/16	25	1365774	238.1	614955	287.4	54.9	−20.7
Quad	2/2/16	2	36405	118.4	30556	129.7	16	−9.5
Mibench	3/2/8	2	20359	64.8	8433	67.2	58.6	−3.7
MVCS	2/3/16	1	31040	119.1	22214	157.8	28.4	−32

an average improvement in the actual implementation area of approximately 42%. However, this area optimization does come at a cost of higher delay.

14.7 Conclusions

This chapter presents a synthesis approach for arithmetic datapaths implemented using a system of polynomial functions. We develop algebraic techniques that efficiently factor coefficients and cubes from the polynomial system, resulting in the generation of linear blocks. Using these blocks as divisors, we perform algebraic division, resulting in a decomposition of the polynomial system. Our decomposition exposes more common terms which can be identified by CSE, leading to a more efficient implementation. Experimental results demonstrate significant area savings using our approach as compared against contemporary datapath synthesis techniques. As part of future work, as datapath designs consume a lot of power, we would like to investigate the use of algebraic transformations in low-power synthesis of arithmetic datapaths.

References

1. Arvind, Shen, X.: Using term rewriting systems to design and verify processors. IEEE Micro **19**(2), 36–46 (1998)
2. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. Ph.D. thesis, Philosophiesche Fakultat an der Leopold-Franzens-Universitat, Austria (1965)
3. Buchberger, B.: A theoretical basis for reduction of polynomials to canonical forms. ACM SIG-SAM Bulletin **10**(3), 19–29 (1976)
4. Chen, Z.: On polynomial functions from $Z_{n1} \times Z_{n2} \times \cdots \times Z_{nr}$ to Z_m. Discrete Mathematics **162**(1–3), 67–76 (1996)
5. Ciesielski, M., Kalla, P., Askar, S.: Taylor expansion diagrams: A canonical representation for verification of dataflow designs. IEEE Transactions on Computers **55**(9), 1188–1201 (2006)
6. Cohen, J.: Computer Algebra and Symbolic Computation. A. K. Peters, Wellesley, MA (2003)
7. DeMicheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill, New York, NY (1994)
8. DeMicheli, G., Sami, M.: Hardware/Software Co-Design. Kluwer, Norwell, MA (1996)
9. Gomez-Prado, D., Ciesielski, M., Guillot, J., Boutillon, E.: Optimizing data flow graphs to minimize hardware implementation. In: Proceedings of Design Automation and Test in Europe, pp. 117–122. Nice, France (2009)
10. Gopalakrishnan, S., Kalla, P.: Optimization of polynomial datapaths using finite ring algebra. ACM Transactions on Design Automation of Electronic System **12**(4), 49 (2007)
11. Gopalakrishnan, S., Kalla, P., Meredith, B., Enescu, F.: Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors. In: Proceedings of the International Conference on Computer Aided Design, pp. 143–148. San Jose, CA (2007)
12. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization. Austin, TX (2001)
13. Hosangadi, A., Fallah, F., Kastner, R.: Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(10), 2012–2022 (2006)

14. JuanCSE: Extensible, programmable and reconfigurable embedded systems group. http://express.ece.ucsb.edu/suif/cse.html
15. Maple. http://www.maplesoft.com
16. Mathews, V.J., Sicuranza, G.L.: Polynomial Signal Processing. Wiley-Interscience, New York (2000)
17. MATLAB/Simulink. http://www.mathworks.com/products/simulink
18. Nicolau, A., Potasman, R.: Incremental tree-height reduction for high-level synthesis. In: Proceedings of the Design Automation Conference. San Francisco, CA (1991)
19. Peymandoust, A., DeMicheli, G.: Application of symbolic computer algebra in high-level data-flow synthesis. IEEE Transactions on CAD **22**(9), 1154–11656 (2003)
20. Smith, J., DeMicheli, G.: Polynomial methods for component matching and verification. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD). San Jose, CA (1998)
21. Smith, J., DeMicheli, G.: Polynomial methods for allocating complex components. In: Proceedings of Design, Automation, and Test in Europe. Munich, Germany (1999)
22. Smith, J., DeMicheli, G.: Polynomial circuit models for component matching in high-level synthesis. IEEE Transactions on VLSI **9**(6), 783–800 (2001)
23. Synopsys: Synopsys Design Compiler and DesignWare library. htpp://www.synopsys.com
24. Verma, A.K., Ienne, P.: Improved use of the Carry-save representation for the synthesis of complex arithmetic circuits. In: Proceedings of the International Conference on Computer Aided Design. San Jose, CA (2004)