Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination

Anup Hosangadi, *Student Member, IEEE*, Farzan Fallah, *Member, IEEE*, and Ryan Kastner, *Member, IEEE*

***Abstract*—Polynomial expressions are frequently encountered in many application domains, particularly in signal processing and computer graphics. Conventional compiler techniques for redundancy elimination such as common subexpression elimination (CSE) are not suited for manipulating polynomial expressions, and designers often resort to hand optimizing these expressions. This paper leverages the algebraic techniques originally developed for multilevel logic synthesis to optimize polynomial expressions by factoring and eliminating common subexpressions. The proposed algorithm was tested on a set of benchmark polynomial expressions where savings of 26.7% in latency and 26.4% in energy consumption were observed for computing these expressions on the StrongARM SA1100 processor core. When these expressions were synthesized in custom hardware, average energy savings of 63.4% for minimum hardware constraints and 24.6% for medium hardware constraints over CSE were observed.**

***Index Terms*—Circuit complexity, common subexpression elimination (CSE), high-level synthesis, polynomials.**

I. Introduction

IN RECENT years, the embedded systems market has seen a huge demand for high-performance and low-power solutions to products such as cellular phones, network routers, and video cameras. This demand has led to the research and development of various high-level optimization techniques for both software [1]–[3] and hardware [4]–[6]. There are a number of optimizations that are well known in the compiler design community [1], [2]; common subexpression elimination (CSE), scalar replacement of aggregates, value numbering, and constant and copy propagation are applied both locally to basic blocks and globally across the whole control flow graph (CFG).

Unfortunately, most of the work that has been done in these compiler tools has been done for general-purpose applications. These techniques do not do a good job of optimizing special functions, such as those consisting of polynomial expressions, which is the focus of this paper.

Polynomial expressions are present in a wide variety of applications domains because any continuous function can be approximated by a polynomial to the desired degree of accuracy [7]–[9]. There are a number of scientific applications that use polynomial interpolation on data of physical phenomenon. Real-time computer graphics applications often use polynomial interpolations for surface and curve generation, texture mapping, etc. [10]. Most DSP algorithms often use trigonometric functions like sine and cosine that can be approximated by their Taylor series as long as the resulting inaccuracy can be tolerated [6], [11]. Furthermore, many adaptive signal processing applications need to use polynomials to model the nonlinearities in high-speed communication channels [12]. Most often, system designers have to depend upon hand-optimized library routines for implementing such functions, which can be both time consuming and error prone. It is important to do a good optimization of polynomial expressions because they are expensive to compute in resource-constrained embedded systems.

This paper presents an algebraic optimization technique for polynomials aimed at reducing the number of operations by algebraic factorization and CSE. This technique can be integrated into the front end of modern optimizing compilers and high-level synthesis frameworks for optimizing programs or data paths consisting of a number of polynomial expressions. As a motivating example for demonstrating the benefits of our technique, consider the polynomial expression shown in Fig. 1(a), which is a multivariate polynomial used in computer

$$P = zu^4 + 4avu^3 + 6bu^2v^2 + 4uv^3w + qv^4$$
(a)

$$d_1 = u^2$$
$$d_2 = v^2$$
$$d_3 = uv$$
$$P = d_1^2 z + 4ad_1d_3 + 6bd_1d_2 + 4wd_2d_3 + qd_2^2$$
(b)

$$P = zu^4 + (4au^3 + (6bu^2 + (4uw + zw)v)v)v$$
(c)

$$d_1 = u^2$$
$$d_2 = 4*u$$
$$P = u(uz + ad_2)d_1 + v^2(u (d_2w + vq) + 6bd_1)$$
(d)

Fig. 1. Optimizations on multivariate quartic-spline polynomial. (a) Unoptimized polynomial. (b) Optimization using CSE. (c) Optimization using Horner transform. (d) Optimization using our algebraic technique.

Manuscript received February 8, 2005; revised July 3, 2005. This work was supported in part by Fujitsu Laboratories of America and by the UC Discovery program under Grant com04-10158. This paper was recommended by Associate Editor L. Stok.

A. Hosangadi and R. Kastner are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106 USA (e-mail: anup@ece.ucsb.edu; kastner@ece.ucsb.edu).

F. Fallah is with Fujitsu Laboratories of America, Sunnyvale, CA 94085 USA (e-mail: farzan.fallah@us.fujitsu.com).

Digital Object Identifier 10.1109/TCAD.2006.875712

graphics for modeling surfaces and textures [10]. This polynomial originally had 23 multiplications. After applying the iterative two-term CSE algorithm [1] on this expression, we get an implementation that has 16 multiplications [Fig. 1(b)]. After applying the Horner form for this expression, we get an implementation with 17 multiplications. But on applying our algebraic method that we develop in this paper, we get an implementation with only 13 multiplications. Such an optimization is impossible to perform using conventional optimization techniques.

Algebraic techniques have been successfully applied in multilevel logic synthesis to reduce the number of literals in a set of Boolean expressions [13]–[16]. These techniques are typically applied to a set of Boolean expressions consisting of hundreds of variables. In this paper, we show that the same techniques can be applied to reduce the number of operations in a set of polynomial expressions. Our algorithms for optimizing polynomial expressions are based on these algebraic techniques.

The rest of this paper is organized as follows. Section II presents some related work on optimizing arithmetic expressions and Boolean expressions. In Section III, we present the transformation of polynomial expressions. In Section IV, we present the algorithms for factorization and CSE. Experimental results are presented in Section V, where we evaluate the benefits of our optimization for execution on an ARM processor and also for custom data path synthesis.

II. Related Work

There have been many early works on the generation of code for arithmetic expressions [17]–[19]. In [17] and [19], the authors propose algorithms for minimizing the number of program steps and storage references in the evaluation of arithmetic expressions given a fixed number of registers. This paper was later extended in [18] to include expressions with common subexpressions. The above works performed optimizations by code generation techniques and did not really present an algorithm that would efficiently reduce the number of operations in a set of arithmetic expressions. Some work was done to optimize code with arithmetic expressions by factorization of the expressions [20]. This paper developed a canonical form for representing the arithmetic expressions and an algorithm for finding common subexpressions. The drawback of this algorithm is that it takes advantage of only the associative and commutative properties of arithmetic operators and therefore can only optimize expressions consisting of only one type of associative and/or commutative operator at a time. As a result, it cannot perform factorization of expressions and generate complex common subexpressions consisting of additions and multiplications.

Horner form is a popular representation of polynomial expressions and is the default way of evaluating polynomial approximations of trigonometric functions in many libraries including the GNU C library [21]. The Horner form of evaluating a polynomial transforms the expression into a sequence of nested additions and multiplications, which are suitable for sequential machine evaluation using multiply accumulate (MAC) instructions. For example, a polynomial in variable x as $p(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_n$ is evaluated as $p(x) = (\cdots((a_0x + a_1)x + a_2)x + \cdots + a_{n-1})x + a_n$. The disadvantage of this technique is that it optimizes only a single polynomial expression at a time and does not look for common subexpressions among a set of polynomial expressions. Furthermore, it is not good at optimizing multivariate polynomial expressions of the type commonly found in computer graphics applications [22].

Symbolic algebra has shown to be a good technique for manipulating polynomials, and it has been shown to have applications in both high-level synthesis [6] and low-power embedded software optimization [11]. The techniques discussed in these works make use of Grobner bases in symbolic algebra to decompose a data flow represented by a polynomial expression to a set of library polynomials. These techniques have been applied in high-level data path synthesis for producing minimum component and minimum latency implementations. The quality of results from these techniques depends heavily on the set of library elements. For example, the decomposition of the polynomial expression $(a^2 - b^2)$ into $(a + b) * (a - b)$ is possible only if there is a library element $(a + b)$ or $(a - b)$. It should be noted that just having an adder in the library will not suffice for this decomposition. Although this problem can be addressed by having a large number of library elements, the run time of the algorithm can become excessively large. This method is useful for the arithmetic decomposition of a data path to library elements, but it is not really useful for simplifying a set of polynomial expressions by factoring and eliminating common subexpressions.

Power optimization techniques for data flow architectures have been explored in detail at the behavioral level [23]–[25], although not particularly for arithmetic expressions. Macii *et al.* [24] and Chandrakasan *et al.* [25] present good surveys of all the relevant works in this area, where different techniques such as operation reduction and operation substitution, power-aware scheduling and resource allocation, multiple-voltage and multiple-frequency scheduling, and bus encoding are explored. Reducing the number of operations is a good technique for reducing the dynamic power because it is directly proportional to the amount of switched capacitance.

Arithmetic expressions have to be computed during address calculation for data transfer intensive (DTI) applications. In [26], different word level optimizing transformations for the synthesis of these expressions such as expression splitting and clustering, induction variable analysis, and common subexpression and factoring are explored. The algebraic techniques that we present in this paper can be used for factoring and eliminating common subexpressions for general polynomial expressions and can be used as one of the transformations in the synthesis of address generators.

Algebraic methods have been successfully applied to the problem of multilevel logic synthesis for minimizing the number of literals in a set of Boolean expressions [13], [14], [16]. These techniques are typically applied to a set of Boolean expressions consisting of thousands of literals and hundreds of variables. Optimization is achieved by decomposition and factorization of the Boolean expressions. There are two main algorithms in the rectangle covering method, namely: 1) Distill

+/-	x	S_3	S_5	S_7
+	1	0	0	0
-	3	1	0	0
+	5	0	1	0
-	7	0	0	1

Fig. 2. Matrix representation of polynomial expressions.

and 2) Condense. The Distill algorithm is preceded by a kernelling algorithm, where a subset of algebraic divisors is generated. Using these divisors, Distill performs multiple-cube decomposition and factorization. This is followed by the Condense algorithm, which performs single-cube decomposition. Our optimization technique for polynomial expressions is based on Distill and Condense algorithms, which we generalized to handle polynomial expressions of any order.

III. Transformation of Polynomial Expressions

The goal of this section is to introduce our matrix representation of arithmetic expressions and their transformations that allow us to perform our optimizations. The transformation is achieved by finding a subset of all possible subexpressions and writing them in matrix form. We also point out the key differences between the algebraic methods for polynomials and those originally developed for multilevel logic synthesis.

A. Representation of Polynomial Expressions

Each polynomial expression is represented by an integer matrix, where there is one row for each product term (cube) and one column for each variable/constant in the matrix. Each element (i, j) in the matrix is a nonnegative integer that represents the exponent of the variable j in the product term i. There is an additional field in each row of the matrix for the sign $(+/-)$ of the corresponding product term. For example, the expression for the Taylor's series approximation for $\sin(x)$ (as shown in Fig. 4) is represented in the matrix form shown in Fig. 2.

B. Definitions

We use the following terminologies to explain our technique. A "literal" is a variable or a constant (e.g., a, b, 2, 3.14...). A "cube" is a product of the variables each raised to a nonnegative integer power. In addition, each cube has a positive or negative sign associated with it. Examples of cubes are $+3a^2b$, $-2a^3b^2c$. An "SOP" representation of a polynomial is the sum of the cubes $(+3a^2b + (-2a^3b^2c) + \cdots)$. An SOP expression is said to be "cube-free" if there is no cube (except for the cube "1") that divides all the cubes of the SOP expression. 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 (cubes). For example, in the expression $P = 3a^2b - 2a^3b^2c$, the expression $P/(a^2b) = (3 - 2abc)$ is a kernel. The cube that is used to obtain a kernel is called a "co-kernel." In the above example, the cube a^2b is a co-kernel. The literals, cubes, kernels, and co-kernels are represented in matrix form in our technique.

C. Algebraic Techniques for Polynomial Expressions and Boolean Expressions in Multilevel Logic Synthesis

The techniques for polynomial expressions that we present in this paper are based on the algebraic methods originally developed for Boolean expressions in multilevel logic synthesis [13]–[15]. We have generalized these methods to handle polynomial expressions of any order and consisting of any number of variables. The changes that were made were mainly because of the differences between Boolean and arithmetic operations. These differences are detailed as follows.

1) Restriction on Single-Cube Containment (SCC): In [14], the Boolean expressions are restricted to be a set of nonredundant cubes such that no cube properly contains another. An example of redundant cubes is in the Boolean expression $f = ab + b$. Here, the cube b can be written as $b = ab + a'b$. Therefore, cube b contains the other cube ab. For polynomial expressions, this notion of cube containment does not exist, and we can have a polynomial $P = ab + b$. Dividing this polynomial by b, $P/b = (ab + b)/b = a + 1$, which we treat as a valid expression. To handle this, we treat "1" as a distinct literal in our representation for polynomial expressions.

For simplifying our algorithm, we place a different restriction on the polynomial expressions. Each polynomial expression is required to be a summation of unequal terms (cubes). Therefore, we cannot have the expression $P = a^2 + ab + ab + b^2$, because there are two instances of the cube ab. If we allow repeated terms in the expressions, then the kernels would also have repeated terms, and there is no way to represent them in the kernel cube matrix (KCM, described in Section III-E). For this expression P, we have the co-kernel and kernel pairs $(a)(a + b + b)$ and $(b)(a + a + b)$. Although this can be handled by modifying our kernel generation algorithm to generate all possible cube-free expressions, this complicates the kernel generation procedure. For handling such expressions, we add up the equal terms before executing the algorithm. Therefore, P is rewritten as $P = a^2 + 2ab + b^2$. This prevents the factorization of the expression as $P = (a + b) * (a + b)$, but it makes the algorithm much simpler.

2) Algebraic Division and the Kernel Generation Algorithm: For Boolean expressions, there is a restriction on the division procedure for obtaining kernels. During division, the quotient and the divisor are required to have orthogonal variable supports. For example, if there is a Boolean expression f, and we need to divide f by another expression g (f/g), then the quotient expression h is defined [14] as "the largest set of cubes in f such that $h \perp g$ (h is orthogonal to g, that is, the variable support of h and the variable support of g are disjoint) and that $hg \in f$." For example, if $f = a(b + c) + d$ and $g = a$, then $f/g = h = (b + c)$. We can see that $h \perp g$ and $hg = a(b + c) \in f$.

For polynomial expressions, we remove this restriction for orthogonality so that we can handle expressions of any order. This is based on one key difference between the multiplication and the Boolean AND operation. For example, the multiplication $a * a = a^2$, but the AND operation $a.a = a$. Therefore, we can have a polynomial $F = a(ab + c) + d$ and a divisor $G = a$. The division $F/G = H = (ab + c)$. We can see that H and G

```
FindKernels({Pi},{Li})
{
    {Pi} = Set of polynomial expressions;
    {Li}  = Set of Literals;
    {Di} = Set of Kernels and Co-Kernels = φ;
    ∀ Expressions Pi  in {Pi}
    {Di} = {Di} ∪ {Kernels(0, Pi , φ)} ∪ {Pi , 1};
     return {Di};
}

Kernels(i,P,d)
{
     i = Literal Number ; P =expression in SOP;
    d  = Cube;

    D = Set of Divisors = φ ;

    for(j = i; j < |L| ; j++)
    {
      If( Lj appears in more than 1 row)
      {
         Fl = Divide (P , Lj );
         C = Largest Cube dividing each cube of Fl;
         if( ( Lk ∉ C)  ∀  (k < j) )
         {
             Fl = Divide(Fl , C); // kernel
             Dl = Merge(d,C, Lj );//  co-kernel

             D = D ∪ Dl ∪ Fl;
            D = D ∪ Kernels(j ,Fl , Dl);
         }
      }
    }
 return D;
}

Divide(P,d)
{
    P = Expression; d = cube ;
    Q = set of rows of P that contain cube d;

    ∀ Rows Ri of Q
    {
        ∀ Columns j in the Row Ri
        Ri[j] = Ri[j] – d[j];
     }
     return Q;
}

Merge(C1, C2, C3)
{
    C1, C2, C3 = cubes ;
    Cube M;
    for(i = 0; i < Number of literals; i++)
       M[i] = C1[i] + C2[i] + C3[i];
   return M;
}
```

Fig. 3. Algorithms for kernel and co-kernel extraction.

are not orthogonal because they share the variable "a". Because the kernel generation algorithm generates kernels by recursive division, we have modified that algorithm to take into account this difference. This algorithm is detailed in Section III-D (Fig. 3).

3) Finding Single-Cube Intersections: In the decomposition and factorization methodology of Boolean expressions presented in [14], multiple-term common factors are extracted first and then single-cube common factors are extracted. For polynomial expressions, we follow the same order. We modify the algorithm for finding single-cube intersections because we have higher order terms. All single-cube common subexpressions can be detected by our method and are described in detail in Sections III-E and IV-B.

D. Generation of Kernels and Co-Kernels

The kernels and co-kernels of polynomial expressions are important because of two reasons. First, all possible minimal algebraic factorizations of an expression can be obtained from the set of kernels and co-kernels of the expression. An algebraic factorization of an expression P is the factorization $P = C * F_1 + F_2$, where C is a cube and F_1 and F_2 are subexpressions consisting of a set of terms. This algebraic factorization is called minimal if the only common cube among the set of terms in F_1 is "1."

Second, all multiple-term (algebraic) common subexpressions can be detected by an intersection among the set of kernels. These two properties are illustrated by the following two theorems.

Theorem 1: All minimal algebraic factorizations of a polynomial expression can be obtained from the set of kernels and co-kernels of the expression.

Proof: Consider the minimal algebraic factorization of $P = C * F_1 + F_2$. By definition, the subexpression F_1 is cube-free because the only cube that commonly divides all the terms of F_1 is "1." We have to prove that F_1 is a part of a kernel expression with the corresponding co-kernel C. Let $\{t_i\}$ be the set of original terms of expression P that covers the terms in F_1. Because F_1 is obtained from $\{t_i\}$, by dividing it by C ($F_1 = \{t_i\}/C$), the common cube of the terms $\{t_i\}$ is C. Let $\{t_j\}$ be the set of original terms of P that also have C as the common cube among them and do not overlap with any of the terms in $\{t_i\}$. Now consider the expression $K_1 = (\{t_i\} + \{t_j\})/C = \{t_i\}/C + \{t_j\}/C = F_1 + \{t'_j\}$, where $\{t'_j\} = \{t_j\}/C$. This expression K_1 is cube-free because we know that F_1 is cube-free. K_1 covers all the terms of the original expression P that contain cube C. Therefore, by definition, K_1 is a kernel of expression P with the cube C as the corresponding co-kernel. Because our kernel generation procedure generates all kernels and co-kernels of the polynomial expressions, the kernel $K_1 = F_1 + \{t'_j\}$ will be generated with the corresponding co-kernel C. Because F_1 is a part of this kernel expression, we proved the theorem. ■

Theorem 2: There is a multiple term common subexpression in the set of polynomial expressions if and only if there is a multiple term intersection among the set of kernels belonging to the expressions.

Explanation: By multiple term intersection, we mean an intersection between kernel expressions yielding two or more terms (cubes). For example, when we intersect the kernels $K_1 = a + bc + d$ and $K_2 = bc + d + f$, then there is a multiple term intersection $= (bc + d)$.

Proof: The "If" case is easy to prove. Let K_1 be the multiple term intersection. It means that there are multiple instances of the subexpression K_1 among the set of expressions. For the

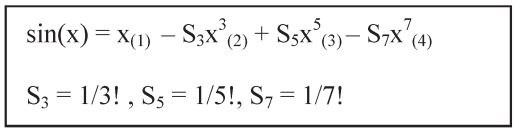

Fig. 4. Evaluation of $\sin(x)$.

"Only If" case, assume that f is a multiple term common subexpression among the set of expressions. If the expression f is cube-free, then f is a part of some kernel expressions as proved in Theorem 1. Each instance of expression f will therefore be a part of some kernel expression, and an intersection among the set of kernels will detect the common subexpression f. If f is not a cube-free expression, then let C be the biggest cube that divides all terms of f, and let $f' = f/C$. The expression f' is now a cube-free expression, and reasoning as above, each instance of f' will be detected by an intersection among the set of kernels.

The algorithm for extracting all kernels and co-kernels of a set of polynomial expressions is shown in Fig. 3. This algorithm is analogous to the kernel generation algorithms in [14] and has been generalized to handle polynomial expressions. The main algorithm Kernels is recursive and is called for each expression in the set of polynomial expressions. The arguments to this function are the literal index i and the cube d, which is the co-kernel extracted in the previous call to the algorithm. The variables in the set of expressions are ordered randomly (the ordering is inconsequential), and the variable index represents the position of the variable in this order.

Before the first call to the Kernels algorithm, the variable index is initialized to "0" and the cube "d" is initialized to φ. The recursive nature of the algorithm extracts kernels and co-kernels within the kernel extracted and returns when there are no kernels present.

The algorithm Divide is used to divide an SOP expression by a cube d. It first collects those rows that contain cube d (those rows R such that all elements $R[i] \geq d[i]$). The cube d is then subtracted from these rows, and these rows form the quotient. The biggest cube dividing all the cubes of an SOP expression is the cube C with the greatest literal count (literal count of C is $\sum_i C[i]$) that is contained in each cube of the expression.

The algorithm Merge is used to find the product of the cubes d, C, and L_j, and is done by adding up the corresponding elements of the cubes. In addition to the kernels generated from the recursive algorithm, the original expression is also added as a kernel with co-kernel "1." Passing the index i to the Kernels algorithm, checking for $L_k \notin C$, and iterating from literal index i to $|L|$ (total number of literals) in the for loop are used to prevent the same kernels from being generated again.

We use the example of the polynomial shown in Fig. 4 to explain our technique. The numbers in the subscripts for each term indicate the term number in the set of expressions. This polynomial is a Taylor's series approximation for $\sin(x)$. The literal set is $\{L\} = \{x, S_3, S_5, S_7\}$. Dividing first by x gives $F_t = (1 - S_3x^2 + S_5x^4 - S_7x^6)$. There is no cube that divides it completely, so we record the first kernel $F_1 = F_t$ with co-kernel x. In the next call to the Kernels algorithm, dividing F by x gives $F_t = -S_3x + S_5x^3 - S_7x^5$.

The biggest cube dividing this completely is $C = x$. Dividing F_t by C gives our next kernel $F_1 = (-S_3 + S_5x^2 - S_7x^4)$ and the co-kernel is $x * x * x = x^3$. In the next iteration, we obtain the kernel $S_5 - S_7x^2$ with co-kernel x^5. Finally, we also record the original expression for $\sin(x)$ with co-kernel "1." The set of all co-kernels and kernels generated is $[x](1 - S_3x^2 + S_5x^4 - S_7x^6)$; $[x^3](-S_3 + S_5x^2 - S_7x^4)$; $[x^5](S_5 - S_7x^2)$; $[1](x - S_3x^3 + S_5x^5 - S_7x^7)$.

E. Constructing KCM

All kernel intersections and multiple-term factors can be identified by arranging the kernels and co-kernels in matrix form. There is a row in the matrix for each kernel (co-kernel) generated and a column for each distinct cube of a kernel generated. The KCM for our $\sin(x)$ expression is shown in Fig. 5. The kernel corresponding to the original expression (with co-kernel "1") is not shown for ease of presentation. An element (i, j) of the KCM is "1" if the product of the co-kernel in row i and the kernel cube in column j gives a product term in the original set of expressions. The number in the parenthesis in each element represents the term number that it represents. A "rectangle" is a set of rows and columns of the KCM such that all the elements are "1." The "value" of a rectangle is the weighed sum of the number of operations saved by selecting the common subexpression or factor corresponding to that rectangle. Selecting the optimal set of common subexpressions and factors is equivalent to finding a maximum valued covering of the KCM and is analogous to the minimum weighed rectangular covering problem described in [15], which is NP-hard. We use a greedy iterative algorithm described in Section IV-A, where we pick the best prime rectangle in each iteration. A "prime rectangle" is a rectangle that is not covered by any other rectangle and thus has more value that any rectangle it covers.

Given a rectangle with the parameters R as the number of rows, $M(R_i)$ as the number of multiplications in row (co-kernel) i, C as the number of columns, $M(C_j)$ as the number of multiplications in column (kernel cube) j, we can calculate its value.

Each element (i, j) in the rectangle represents a product term equal to the product of co-kernel i and kernel cube j, which has a total number of $M(R_i) + M(C_j) + 1$ multiplications. The total number of multiplications represented by the whole rectangle is equal to $R * \sum_C M(C_j) + C * \sum_R M(R_i) + R * C$. Each row in the rectangle has $C - 1$ additions for a total of $R * (C - 1)$ additions. By selecting the rectangle, we extract a common factor with $\sum_C M(C_j)$ multiplications and $C - 1$ additions. This common factor is multiplied by each row, which leads to a further $\sum_R M(R_i) + R$ multiplications. The value of the rectangle can be described as the weighed sum of the savings in the number of multiplications and additions and is given by

$$\begin{aligned}\text{Value}_1 = m * \Bigg\{ & (C-1) * \left(R + \sum_R M(R_i) \right) \\ & + (R-1) * \left(\sum_C M(C_j) \right) \Bigg\} \\ & + (R-1) * (C-1). \end{aligned} \tag{1}$$

		1	2	3	4	5	6	7	8	9
		1	**$-S_3x^2$**	**S_5x^4**	**$-S_7x^6$**	**$-S_3$**	**S_5x^2**	**$-S_7x^4$**	**S_5**	**$-S_7x^2$**
1	**x**	$1_{(1)\_}$	$1_{(2)\_}$	$1_{(3)}$	$1_{(4)}$					
2	**x^3**					$1_{(2)}$	$1_{(3)}$	$1_{(4)}$		
3	**x^5**								$1_{(3)\_}$	$1_{(4)}$

Fig. 5. KCM for $\sin(x)$ example.

The weighing factor m can be selected by the designer depending on the relative cost of multiplication and addition on the target architecture. For example, multiplication takes about five times more cycle time than addition on an ARM processor [27]. For custom hardware synthesis, multiplication has a much higher power consumption compared to addition. In [28], the average energy consumption of a multiplier is about 40 times that of an adder at 5 V.

F. Constructing Cube Literal Incidence Matrix (CIM)

KCM allows the detection of multiple cube common subexpressions and factors. All possible cube intersections can be identified by arranging the cubes in a matrix, where there is a row for each cube and a column for each literal in the set of expressions. The CIM for our $\sin(x)$ expression is shown in Fig. 2.

Each cube intersection appears in the CIM as a rectangle. A "rectangle" in the CIM is a set of rows and columns such that all the elements are nonzero. This CIM provides a canonical representation of the cubes of the expression, which enables the detection of all single-cube common subexpressions, as is illustrated by the following theorem.

Theorem 3: There is a single-cube common subexpression in the set of polynomial expressions if and only if there is a rectangle with more than one row in the cube intersection matrix (CIM).

Proof: For the "If" case, assume that there is a rectangle with more than one row in the CIM. The elements of this rectangle are nonzero by definition. Let C be the biggest common cube that is obtained from the minimum values in each column of the rectangle. This cube C is the single-cube common subexpression. For the "Only If" case, assume that there is a common cube D in the set of expressions. Let $\{v_d\}$ be the set of variables that cube D is made of. There will be nonzero values in the columns corresponding to the variables $\{v_d\}$ for each term that contains cube D. As a result, a rectangle containing the columns corresponding to the variables $\{v_d\}$ will be formed. Thus, the theorem is proved. ■

The value of a rectangle is the number of multiplications saved by selecting the single-term common subexpression corresponding to that rectangle. The best set of common cube intersections is obtained by a maximum-valued covering of the CIM. We use a greedy iterative algorithm described in Section III-B, where we select the best prime rectangle in each iteration. The common cube C corresponding to the prime rectangle is obtained by finding the minimum value in each column of the rectangle. The value of a rectangle with R rows can be calculated as follows.

Let $\sum C[i]$ be the sum of integer powers in the extracted cube C. This cube saves $\sum C[i] - 1$ multiplications in each row of the rectangle. The cube itself needs $(\sum C[i] - 1)$ multiplications to compute. Therefore, the value of the rectangle is given by

$$\text{Value}_2 = (R-1) * \left(\sum C[i] - 1\right). \tag{2}$$

```
FindKernelIntersections( {P_i}, {L_ij})
{
    while(1)
    {
      D = FindKernels({P_i}, {L_ij});
      KCM = Form Kernel Cube Matrix (D);
      {R} =Set of new kernel intersections = φ ;
      {V} =Set of new variables = φ;
      if (no favorable rectangle) return;
      while(favorable rectangles exist)
      {
        {R} = {R} ∪ Best Prime Rectangle;
        {V} = {V} ∪ New Literal;
        Update KCM;
      }
      Rewrite {P_i} using{R};
      {P_i} = {P_i} ∪ {R};
      {L_ij} = {L_ij} ∪ {V};
    }
}
```

Fig. 6. Algorithm for finding kernel intersections.

IV. Factoring and Eliminating Common Subexpressions

In this section, we present two algorithms that detect kernel and cube intersections. We first extract kernel intersections and eliminate all multiple-term common subexpressions and factors. We then extract cube intersections from the modified set of expressions and eliminate all single-cube common subexpressions.

A. Extracting Kernel Intersections

The algorithm for finding kernel intersections is shown in Fig. 6 and is analogous to the Distill procedure [14] in logic synthesis. It is a greedy iterative algorithm in which the best prime rectangle is extracted in each iteration. In the outer loop, kernels and co-kernels are extracted for the set of expressions $\{p_i\}$, and KCM is formed from that. The outer loop exits if there is no favorable rectangle in the KCM. Each iteration in the inner loop selects the most valuable rectangle, if present, based on our value function in (1).

		1	2	3	4
		1	x^2d^1	S_5	$-S_7x^2$
1	x	$1_{(1)}$	$1_{(2)}$		
2	x^2			$1_{(4)}$	$1_{(5)}$

Fig. 7. KCM (second iteration).

$\sin(x) = (xd_3)_{(1)}$
$d_3 = (1)_{(2)} + (x^2d_1)_{(3)}$
$d_2 = (S_5)_{(4)} - (x^2S_7)_{(5)}$
$d_1 = (x^2d_2)_{(6)} - (S_3)_{(7)}$

Fig. 8. Expressions after kernel intersection.

This rectangle is added to the set of expressions, and a new literal is introduced to represent this rectangle. The KCM is then updated by removing those 1s in the matrix that correspond to the terms covered by the selected rectangle. For example, for the KCM for the $\sin(x)$ expression shown in Fig. 5, consider the rectangle that is formed by the row $\{2\}$ and the columns $\{5, 6, 7\}$. This is a prime rectangle having $R = 1$ row and $C = 3$ columns. The total number of multiplications in rows $\sum_R M(R_i)$ is 2 and total number of multiplications in the columns $\sum_C M(C_i)$ is 6. Using the value function in (1), the value of this rectangle is $6 * m$ $(m \geq 1)$. This is the most valuable rectangle and is selected. Because this rectangle covers the terms 2, 3, and 4, the elements in the matrix corresponding to these terms are deleted. No more favorable (valuable) rectangles are extracted in this KCM, and now we have two expressions, after rewriting the original expression for $\sin(x)$, i.e.

$$\begin{aligned}\sin(x) &= x_{(1)} + (x^3 d_1)_{(2)} \\ d_1 &= (-S_3)_{(3)} + (S_5 x^2)_{(4)} - (S_7 x^4)_{(5)}.\end{aligned}$$

Extracting kernels and co-kernels as before, we have the KCM shown in Fig. 7. Again, the original expressions for $\sin(x)$ and d_1 are not included in the matrix to simplify representation. From this matrix, we can extract two favorable rectangles corresponding to $d_2 = (S_5 - S_7x^2)$ and $d_3 = (1 + x^2d_1)$. No more rectangles are extracted in the subsequent iterations, and the set of expressions can now be written as shown in Fig. 8.

B. Extracting Cube Intersections

The Distill algorithm discussed in the previous section could find only multiple-term (two or more) common subexpressions. We need an algorithm that can find single-term common subexpressions. We do this optimization by means of a CIM, which was discussed in Section III-F. Single-term common subexpressions appear as rectangles in this matrix, where the rectangle entries can have any nonzero integers. The optimization is performed by iteratively eliminating the rectangle having the most savings. For example, consider the three terms $F_1 = a^2b^3c$, $F_2 = a^3b^2$, and $F_3 = b^2c^3$. The CIM for these expressions is shown in Fig. 9(a).

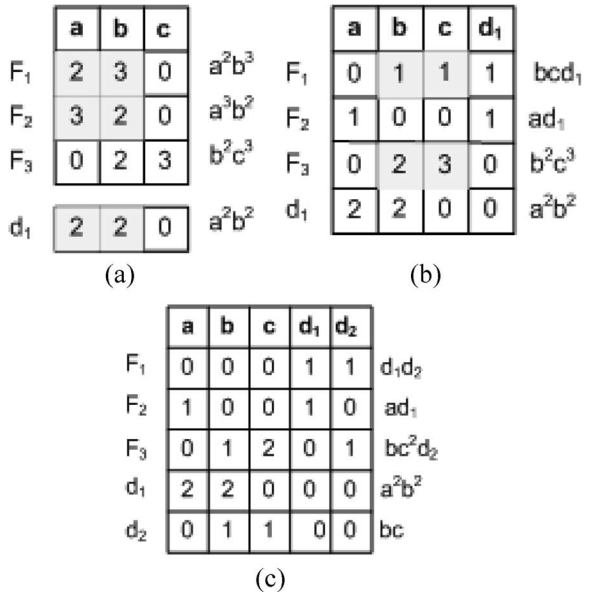

Fig. 9. Extracting single-term common subexpressions.

A rectangle with rows corresponding to F_1 and F_2 and columns corresponding to the variables a and b yields the common subexpression $d_1 = a^2b^2$. Rewriting the CIM, we get the matrix shown in Fig. 9(b). Another rectangle with rows corresponding to F_1 and F_3 and columns corresponding to the variables b and c is obtained, eliminating the common subexpression $d_2 = b * c$. After rewriting, we get the CIM as shown in Fig. 9(c). No more favorable rectangles can be found in this matrix, and we can stop here.

The algorithm for finding the cube intersection is done at the end of the Distill algorithm and is shown in Fig. 10. In each iteration of the inner loop, the best prime rectangle is extracted using the value function in (2). The cube intersection corresponding to this rectangle is extracted from the minimum value in each column of the rectangle. After each iteration of the inner loop, the CIM is updated by subtracting the extracted cube from all rows in the CIM in which it is contained (Collapse procedure). Coming back to our $\sin(x)$ example, consider the CIM in Fig. 11 for the set of expressions in Fig. 8. The prime rectangle consisting of the rows $\{3, 5, 6\}$ and the column $\{1\}$ is extracted, which corresponds to the common subexpression $C = x^2$. Using (2), we can see that this rectangle saves two multiplications. This is the most favorable rectangle and is chosen. No more cube intersections are detected in the subsequent iterations. A literal $d_4 = x^2$ is added to the expressions that can now be written as in Fig. 12.

C. Complexity and Quality of Presented Algorithms

The complexity of the kernel generation algorithm (Fig. 3) depends on the number of variables, the number of terms, the order of the expressions, and the distribution of the variables in the original terms. We do not provide an analysis of the worst case for the kernel generation algorithm. In each iteration, most of the time is spent in enumerating all the rectangles to find the most valuable rectangle. In the worst case, the kernel intersection matrix (KIM) can have a structure like the one

```
FindCubeIntersections( {Pi}, {Li})
{
    while(1)
    {
        M = Cube Literal incidence Matrix
        {V} = Set of new variables = φ;
        {C} = Set of new cube intersections = φ;
        if( no favorable rectangle present) return;

        while(Favorable rectangles exist)
        {
            Find B = Best Prime Rectangle;
            {C} = {C} ∪ Cube corresponding to B;
            {V} = {V} ∪ New Literal;
            Collapse(M,B);
        }
        M = M ∪ {C};
        {Li} = {Li} ∪ {V};
    }
}
Collapse(M,B)
{
    ∀ rows i of M that contain cube B
      ∀ columns j of B
      M[i][j] = M[i][j] – B[j];
}
```

Fig. 10. Algorithm for finding cube intersections.

Term	+/-	1	2	3	4	5	6	7	8
		x	d_1	d_2	d_3	1	S_3	S_5	S_7
1	+	1	0	0	1	0	0	0	0
2	+	0	0	0	0	1	0	0	0
3	+	2	1	0	0	0	0	0	0
4	+	0	0	0	0	0	0	1	0
5	-	2	0	0	0	0	0	0	1
6	+	2	0	1	0	0	0	0	0
7	-	0	0	0	0	0	1	0	0

Fig. 11. CIM for expressions in Fig. 8.

$$d_4 = x*x$$
$$d_2 = S_5 - S_7*d_4$$
$$d_1 = d_2*d_4 - S_3$$
$$d_3 = d_1*d_4 + 1$$
$$\sin(x) = x*d_3$$

Fig. 12. Final optimization of $\sin(x)$ example.

shown for the square matrix in Fig. 13, where all elements, except the ones on the diagonal, are nonzero.

In such a scenario, the number of rectangles generated is exponential in the number of rows/columns in the matrix. The number of rows in the matrix is equal to the number of kernels that are generated, and the number of columns is equal to the number of distinct kernel cubes. The algorithm for finding

	1	1	1	1
1		1	1	1
1	1		1	1
1	1	1		1
1	1	1	1	

Fig. 13. Worst case complexity for finding kernel intersections.

single-cube intersections also has a worst case exponential complexity when the CIM is of the form shown in Fig. 13. The number of rows of the CIM is equal to the number of terms, and the number of columns is equal to the number of literals in the set of expressions.

The algorithms that we presented are greedy heuristic algorithms. To the best of our knowledge, there has been no previous work done for finding an optimal solution to the CSE problem. An optimal solution can be generated by a brute force exploration of the entire search space where all possible subexpressions in different selection orderings are explored. Because this is impractical for even moderately sized examples, we do not compare our solution with the optimal one. For our greedy heuristic, the average run time for the examples that we investigated is only 0.45 s.

V. Experimental Results

The goal of our experiments was to investigate the usefulness of our technique in reducing the number of operations in polynomial expressions occurring in some real applications. We found polynomial expressions to be prevalent in signal processing [29] and in three-dimensional (3-D) computer graphics [10]. We optimized the polynomials using different methods, CSE, Horner form, and our algebraic technique. We used the Jouletrack simulator [30] to estimate the reduction in latency and energy consumption for computing the polynomials on the StrongARM SA1100 microprocessor. We used the same set of randomly generated inputs for simulating the different programs. Table I shows the result of our experiments, where we compare the savings in the number of operations produced by our method over CSE and the Horner form. The first two examples are obtained from source codes from signal processing applications [29], where the polynomials are obtained by approximating trigonometric functions by their Taylor series expansions. The next four examples are multivariate polynomials obtained from 3-D computer graphics [22]. The results show that our optimizations reduce the number of multiplications on an average by 34% over CSE and by 34.9% over Horner. The number of additions is the same in all examples. This is because all the savings are produced by efficient factorization and elimination of single-term common subexpressions, which only reduce the number of multiplications. The average run time for our technique for these examples was only 0.45 s.

The results also show the effect of simulation on the StrongARM SA1100 processor, where we observe an average latency reduction of 26.7% over CSE and 26.1% over the Horner scheme. We also observed an energy reduction of 26.4% over CSE and 26.3% over the Horner scheme. The

TABLE I
COMPARING NUMBER OF ADDITIONS (A) AND MULTIPLICATIONS (M) PRODUCED BY DIFFERENT METHODS

	Application	Function	Unoptimized		Using CSE		Using Horner		Using our Algebraic Technique		Reduction in Latency and Energy consumption for execution on StrongARM SA1100			
											CSE		Horner	
			A	M	A	M	A	M	A	M	Latency (%)	Energy (%)	Latency (%)	Energy (%)
1	Fast Convolution	FFT	7	56	7	20	7	30	7	10	26.8	26.1	44.0	42.3
2	Gaussian noise filter	FIR	6	34	6	23	6	20	6	13	30.5	26.4	16.5	16.1
3	Graphics	quartic-spline	4	23	4	16	4	17	4	13	7.0	10.7	35.5	35.1
4	Graphics	quintic-spline	5	34	5	22	5	23	5	16	23.4	24.8	23.3	26.2
5	Graphics	chebyshev	8	32	8	18	8	18	8	11	33.4	31.3	27.5	25.4
6	Graphics	cosine-wavelet	17	43	17	23	17	20	17	17	39.1	39.0	9.7	12.8
	Average		**7.8**	**37**	**7.8**	**20.3**	**7.8**	**21.3**	**7.8**	**13.3**	**26.7**	**26.4**	**26.1**	**26.3**

TABLE II
SYNTHESIS RESULTS WITH MINIMUM HARDWARE CONSTRAINTS

	Area (%) (I)		Energy @5V (%) (II)		Energy Delay @5V (%) (III)		Energy (scaled voltage) (%) (IV)	
	C	H	C	H	C	H	C	H
1	18.6	6.5	33.5	69.9	58.4	90.8	88.9	99.0
2	7.5	0.1	13.6	25.6	20.4	39.4	24.6	49.5
3	0.3	-4.2	21.6	29.3	39.0	48.8	52.2	64.6
4	-7.5	-24.2	29.4	10.4	47.6	25.9	62.2	36.9
5	5.6	2.5	37.0	28.7	57.1	46.1	74.3	59.8
6	3.7	2.0	44.8	36.8	62.8	54.8	78.3	69.7
Average	**4.7**	**-2.8**	**30.0**	**33.4**	**47.5**	**50.9**	**63.4**	**63.2**

energy consumption measure by the simulator [30] is only for the processor core.

We synthesized the polynomials optimized by CSE, Horner, and our algebraic method to observe the improvements for hardware implementation. We synthesized the polynomials using Synopsys Behavioral Compiler and Synopsys Design Compiler using the 1.0 μ power2_sample.db technology library, with a clock period of 40 ns, operating at 5 V. We used this library because it was the only one we had that was characterized for power consumption. We used the Synopsys DesignWare library for the functional units. The adder and multiplier in this library took one clock cycle and two clock cycles, respectively, at this clock period. We synthesized the designs with both minimum hardware constraints and medium hardware constraints. We then interfaced the Synopsys Power Compiler with the Verilog RTL simulator VCS to capture the total power consumption including switching, short circuit, and leakage power. All the polynomials were simulated with the same set of randomly generated inputs.

Table II shows the synthesis results when the polynomials were scheduled with minimum hardware constraints. Typically, the hardware allocated consisted of a single adder, a single multiplier, registers, and control units. We measured the reduction in area, energy, and energy-delay product of the polynomials optimized using our technique over the polynomials optimized using CSE (C) and Horner transform (H). The results show that there is not much difference in area for the different implementations, but there is a significant reduction in total energy consumption (average 30% over CSE and 33.4% over Horner) and energy delay product (average 47.5% over CSE and 50.9% over Horner). Because our technique gave significantly reduced latency, we estimated the savings in energy consumption for the same latency using voltage scaling.

We obtained the scaled down voltage for the new latency using the library parameters and estimated the new energy consumption using the quadratic relationship between the energy consumption and the supply voltage. This voltage scaling was done by comparing the latencies obtained by our technique with that of CSE and Horner separately.

The results (Column IV) show an energy reduction of 63.4% over CSE (C) and 63.2% over Horner (H) with voltage scaling. Table III shows the synthesis results for medium hardware constraints. Medium hardware implementations tradeoff area for energy efficiency. Although they have a larger area, they have a shorter schedule and lesser energy consumption compared to those produced from minimum hardware constraints. We constrained the Synopsys Behavioral Compiler to allocate a maximum of four multipliers for each example. Table IV shows the number of adders (A) and multipliers (M) allocated for the different implementations. The scheduler will allocate fewer multipliers than four if the same latency can be achieved by the fewer number of resources.

TABLE III
SYNTHESIS RESULTS WITH MEDIUM HARDWARE CONSTRAINTS

	Area (%)		Energy @5V (%)		Energy Delay @5V (%)	
	C	H	C	H	C	H
1	44.0	48.0	9.8	63.9	-12.7	81.0
2	30.5	3.9	16.1	39.2	9.7	44.1
3	14.8	1.0	9.7	29.6	20.3	58.7
4	8.3	3.7	42.5	29.1	44.9	37.0
5	8.9	9.0	28.2	29.5	39.5	40.6
6	8.0	6.6	41.4	40.8	58.4	59.7
Average	**19.0**	**12.0**	**24.6**	**38.7**	**26.7**	**53.5**

TABLE IV
MULTIPLIERS (M) AND ADDERS (A) ALLOTTED WITH MEDIUM HARDWARE CONSTRAINTS

	CSE		Horner		Our Technique	
	M	A	M	A	M	A
1	4	2	4	1	2	1
2	3	1	2	1	2	1
3	4	1	4	1	4	1
4	3	1	3	2	3	2
5	4	1	4	2	4	2
6	3	1	3	1	3	2

The results show a significant reduction in total energy consumption (average 24.6% over CSE (C) and 38.7% over the Horner (H) scheme) as well as energy delay product (average 26.7% over CSE and 53.5% over Horner). We obtain reduction in energy delay product for every example except for the first one, where the energy delay product for the expression synthesized using CSE is 12.7% less than that produced by our method. This is because the latency of the CSE-optimized expression when four multipliers are used is much less compared to the one optimized using our algebraic method (which uses two multipliers). The delay for the Horner scheme is much worse than both CSE and our technique because the nested additions and multiplications of the Horner scheme typically result in a longer critical path.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their detailed feedback. They would also like to thank T. Sidle at Fujitsu Laboratories of America for his assistance.

REFERENCES

[1] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.
[2] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. San Mateo, CA: Morgan Kaufmann, 2002.
[3] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 6, pp. 801–804, Dec. 2001.
[4] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[5] H. De Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six, and L. Claesen, "CATHEDRAL-II—A computer-aided synthesis system for digital signal processing VLSI systems," *Comput.-Aided Eng. J.*, vol. 5, no. 2, pp. 55–66, Apr. 1988.
[6] A. Peymandoust and D. Micheli, "Using symbolic algebra in algorithmic level DSP synthesis," in *Proc. Design Automation Conf.*, 2001, pp. 277–282.
[7] D. E. Knuth, "The art of computer programming," in *Seminumerical Algorithms*, 2nd ed, vol. 2. Reading, MA: Addison-Wesley, 1981.
[8] C. Fike, *Computer Evaluation of Mathematical Functions*. Englewood Cliffs, NJ: Prentice-Hall, 1968.
[9] J. Villalba, G. Bandera, M. A. Gonzalez, J. Hormigo, and E. L. Zapata, "Polynomial evaluation on multimedia processors," in *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures Processors*, 2002, pp. 265–274.
[10] R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. San Mateo, CA: Morgan Kaufmann, 1987.
[11] A. Peymandoust, T. Simunic, and G. D. Micheli, "Low power embedded software optimization using symbolic algebra," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 8, pp. 964–975, Aug. 2003.
[12] V. J. Mathews, "Adaptive polynomial filters," *IEEE Signal Process. Mag.*, vol. 8, no. 3, pp. 10–26, Jul. 1991.
[13] A. S. Vincentelli, A. Wang, R. K. Brayton, and R. Rudell, "MIS: Multiple level logic optimization system," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 6, no. 6, pp. 1062–1081, Nov. 1987.
[14] R. K. Brayton and C. T. McMullen, "The decomposition and factorization of boolean expressions," in *Proc. Int. Symp. Circuits Systems*, May 1982, pp. 49–54.
[15] R. K. Brayton, R. Rudell, A. S. Vincentelli, and A. Wang, "Multi-level logic optimization and the rectangular covering problem," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1987, pp. 66–69.
[16] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of boolean expressions," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 11, no. 6, pp. 778–793, Jun. 1992.
[17] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM*, vol. 23, no. 3, pp. 488–501, Jul. 1976.
[18] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *ACM*, vol. 24, no. 1, pp. 146–160, Jan. 1977.
[19] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *Commun. ACM*, vol. 17, no. 4, pp. 715–728, Oct. 1970.
[20] M. A. Breuer, "Generation of optimal code for expressions via factorization," *Commun. ACM*, vol. 12, no. 6, pp. 333–340, Jun. 1969.
[21] GNU C Library, [Online]. Available: http://www.gnu.org/software/libc
[22] G. Nurnberger, J. W. Schmidt, and G. Walz, *Multivariate Approximation and Splines*. New York: Springer-Verlag, 1997.
[23] J.-M. Chang and M. Pedram, "Energy minimization using multiple supply voltages," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 5, no. 4, pp. 436–443, Dec. 1997.
[24] E. Macii, M. Pedram, and F. Somenzi, "High-level power modeling, estimation, and optimization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 17, no. 11, pp. 1061–1079, Nov. 1998.
[25] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Optimizing power using transformations," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 1, pp. 12–31, Jan. 1995.
[26] M. A. Miranda, F. V. M. Catthoor, M. Janssen, and H. J. De Man, "High-level address optimization and synthesis techniques for data-transfer-intensive applications," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 6, no. 4, pp. 677–686, Dec. 1998.
[27] ARM7TDMI Tech Reference Manual. (2003). [Online]. Available: http://www.arm.com/documentation/ARMProcessor_Cores/index.html
[28] V. Krishna, N. Ranganathan, and N. Vijayakrishnan, "An energy efficient scheduling scheme for signal processing applications," in *Proc. Asilomar Conf. Signals, Syst., Comput.*, 1998, vol. 2, pp. 1057–1061.
[29] P. M. Embree, *C Algorithms for Real-Time DSP*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
[30] A. Sinha and A. P. Chandrakasan, "Jouletrack-A web based tool for software energy profiling," in *Proc. Design Automation Conf.*, 2001, pp. 220–225.

Anup Hosangadi (S'03) received the B.E. degree in electrical and electronics engineering from the National Institute of Technology, Trichy, India, in 2001, and the M.S. degree in computer engineering from the University of California, Santa Barbara, in 2003. He is currently working toward the Ph.D. degree at the University of California, Santa Barbara.

His research interests include high-level synthesis, combinatorial optimization, and computer arithmetic.

Mr. Hosangadi was the recipient of the Best Student Paper Award at the International Conference on VLSI Design.

Farzan Fallah (S'97–M'99) received the B.S. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 1992, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1996 and 1999, respectively.

In April 1999, he joined Fujitsu Laboratories of America, where he is currently the leader of the low-power design project. He has authored and coauthored over 40 papers. His primary research interests include low-power design and verification.

Dr. Fallah is a member of the Association for Computing Machinery and the IEEE Management Society. He has served on the technical program committee of DATE, HLDVT, ISQED, and the Ph.D. Forum at DAC and has initiated the Ph.D. Forum at ASP-DAC. He has received a number of awards including a Design Automation Conference Best Paper Award and an International Conference on VLSI Design Best Paper Award.

Ryan Kastner (S'00–M'02) received the B.S. and M.S. degrees in electrical engineering and computer engineering from Northwestern University, Evanston, IL, in 1999 and 2000, respectively, and the Ph.D. degree in computer science from the University of California, Los Angeles, in 2002.

He is an Assistant Professor in the Department of Electrical and Computer Engineering, University of California, Santa Barbara. He has published over 60 technical articles and is the author of the book *Synthesis Techniques and Optimizations for Reconfigurable Systems* (Kluwer, 2003). His current research interests include the realm of embedded system design, in particular, using reconfigurable computing for digital signal processing in applications such as wireless communications and underwater sensor networking.

Dr. Kastner is a member of numerous conference technical committees including GLOBECOM, Design Automation Conference, International Conference on Computer Design, Great Lakes Symposium on VLSI, the International Conference on Engineering of Reconfigurable Systems and Algorithms, and the International Symposium on Circuits and Systems. He serves on the editorial board for the *Journal of Embedded Computing*.