Energy Efficient Hardware Synthesis of Polynomial Expressions

Anup Hosangadi
University of California, Santa Barbara
anup@ece.ucsb.edu

Farzan Fallah
Fujitsu Labs of America, Inc.
farzan@fla.fujitsu.com

Ryan Kastner
University of California, Santa Barbara
kastner@ece.ucsb.edu

Abstract

Polynomial expressions are used to approximate a wide variety of functions commonly found in signal processing and computer graphics applications. Computing these polynomial expressions in hardware consumes a lot of energy and therefore careful optimization of these expressions is important in order to achieve low energy consumption. Unfortunately, current optimization techniques for reducing complexity of expressions such as Common Subexpression Elimination (CSE) cannot do a good optimization. In this paper, we present an algebraic technique to reduce the energy consumption of custom datapath implementation of polynomials by reducing the number of energy intensive operations. Our techniques can handle polynomial expressions of any order and containing any number of variables. Synthesis of a set of benchmark polynomials verified the advantages of our technique in reducing energy consumption, where we observed up to ***58%*** *improvement over CSE.*

1. Introduction

Portable embedded systems typically have stringent constraints on total power consumption. The rapid growth of portable consumer electronics such as camcorders, mobile phones and handheld devices has pushed the research towards low power, high performance designs. Implementing frequently used computation intensive functions in hardware is a good solution to meet these objectives. Advance in high level synthesis methodologies have made it possible to build low cost Application Specific Integrated Circuits (ASICs) for implementing such functions. A lot of embedded system applications have significant data-flow segments [1], which are good candidates for custom hardware implementation. Developing automatic techniques that can fully optimize these computations is very important to achieve low energy consumption and meet time to market demands. Unfortunately, most high level synthesis tools provide limited support for reducing the complexity of large datapaths like polynomial expressions which are prevalent in many embedded system applications. Therefore developing application specific techniques is important in order to achieve low power consumption.

Switching power is the major component of active power dissipation in a system and is directly proportional to the number of operations, given a fixed supply voltage. Most polynomial expressions have a large number of multiplications, which are power and area hungry. In [2] the average energy consumption of a multiplier was reported to be 40 times more than that of an adder at 5 V. Therefore careful optimization of these polynomial expressions can have significant impact on the total energy consumption in the system.

Polynomial expressions are common in signal processing applications since any continuous function can be approximated by a polynomial to the desired degree of accuracy [1, 3]. Real time computer graphics applications often use polynomial interpolations for surface and curve generation, texture mapping etc. [4]. Consider the set of polynomial expressions shown in Figure 1a. The subscripts under each term represent the term numbers in the set of expressions. Using an iterative common subexpression elimination algorithm [6] on an intermediate representation of these polynomials results in the set of expressions shown in Figure 1b which has 12 multiplications and four addition/subtractions? The Horner form of the expressions was obtained by using the convert function in Maple [7], with the variable set assigned to {x,y,z,4}. These expressions have a total of 12 multiplications and four additions/subtractions. Using our algebraic technique on the original unoptimized set of polynomials we get the set of expressions shown in Figure 1d. These set of expressions have a total of seven multiplications and three additions/subtractions. Normalizing the energy consumption unit to that of an addition and assuming a multiplication to consume 40 times more energy than an addition/subtraction, polynomials optimized using the algebraic technique would have an energy savings of 41.5% over both CSE and Horner and 56% over the original unoptimized case.

In this paper we present a technique for the algebraic optimization of polynomial expressions with a view to minimizing energy consumption of the custom datapath for these expressions. This technique is based on the algebraic methods originally developed for multi-level logic synthesis. We measure the actual improvement in performance and energy by synthesizing and simulating these complex datapaths using Synopsys Design Compiler™ and Power Compiler™. The rest of the paper is organized as follows: Section 2 presents some related work on power optimization transforms. We present our technique for algebraic optimization of polynomial expressions in Section 3. Experimental results are presented in Section 4. Finally we conclude and talk about future work in Section 5.

2. Related work

Common subexpression elimination (CSE) and value numbering are conventionally used to reduce the number of operations in a data-flow segment, by iteratively finding and eliminating two term common subexpressions [6]. But they are not suitable for optimizing polynomial expressions since they cannot perform factorization, and

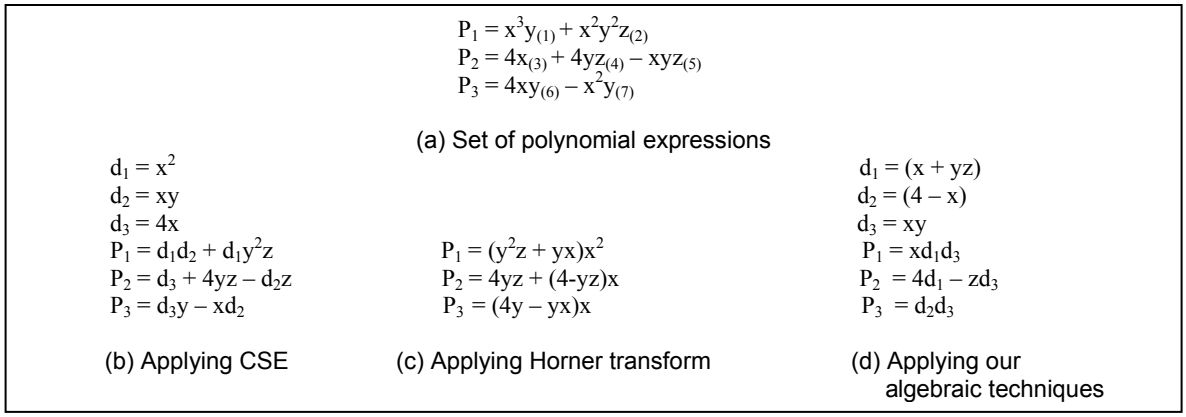

Figure 1. Example showing the superiority of our technique over CSE and Horner form

cannot extract common subexpression having more than two operands at a time.

A number of transformations have been proposed in [8] to evaluate alternative architectures and select the one that results in the lowest power consumption. The authors have proposed operation reduction as one of the transformations since it directly reduces the amount of switching capacitance. They have used the Horner transform to evaluate the polynomials in various signal processing applications such as Finite Fourier Transform (FFT) and Discrete Cosine Transform (DCT). A Horner form equation turns a polynomial expression into a nested set of multiplies and adds, perfect for machine evaluation using multiply-add instructions.

A polynomial in a variable x ,$p(x) = a_0x^n + a_1x^{n-1} + \ldots\ldots + a_{n-1}x + a_n$ is evaluated as

$p(x) = (\ldots((a_0x + a_1)x + a_2)x + \ldots a_{n-1})x + a_n$.

But the Horner form does look for reducing common subexpressions, and also is not effective in optimizing multiple variable polynomial expressions.

Symbolic algebra has been shown to be a powerful technique for manipulating polynomials, and has been used for low power embedded software optimization [3], where the behavioral C code for an MP3 decoder has been mapped to library routines optimized for power. These techniques can extract non-trivial factors like $a^2 - b^2 = (a+b)(a-b)$, which our algebraic technique cannot. However a major drawback of the symbolic algebra routines is that they are heavily dependant on the library elements. In the above example the factorization will be possible only if there are the library elements (a+b) or (a-b). It should be noted that this requirement is different than having an adder in the library. Besides these techniques are good at optimizing only one polynomial expression at a time, whereas our technique can optimize multiple polynomial expressions at the same time to get better results.

3. Algebraic optimization

The goal of this section is to show our representation of polynomial expressions and present algorithms for the reduction of energy intensive operations in the set of polynomials.

3.1 Preliminaries

We represent each polynomial expression using an integer matrix, where there is one row for each product term and one column for each variable/constant in the set of expressions. Each element (i,j) in the matrix is a non-negative integer that represents the exponent of the variable j in the term i. We use the following terminology for explaining 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 non-negative integer power. In addition each cube has a positive or a 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 ($+3a^2b + (-2a^3b^2c) + \ldots$). An SOP expression is said to be **cube-free** if there is no literal or cube 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. 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.

3.2 Generation of kernels and co-kernels

The importance of kernels is illustrated by the following theorem [9]:

Theorem: Two expressions f and g have a common multiple cube divisor if and only if there are two kernels K_f and K_g belonging to the set of kernels generated for f and g respectively having a multiple cube intersection.

Therefore by finding the set of all kernels and their intersections, we can find all the multiple cube intersections of the expressions. The algorithm for extracting all kernels and co-kernels of a set of arithmetic expressions $\{P_i\}$ is shown in Figure 2. This algorithm is based on the algorithm in [9] and has been modified to handle polynomial expressions. The recursive algorithm **Kernels** is called for each expression with the literal index 0 and the cube d which is the co-kernel extracted up to this point, initialized to φ. The recursive nature of the algorithm extracts kernels and co-kernels within the kernel extracted and returns when there are no more 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.

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

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( L_j appears in more than 1 row)
     {
        F_t = Divide (P , L_j);
        C = Largest Cube dividing each cube of F_t;
        if( ( L_k ∉ C) ∀ (k < j) )
        {
            F_1 = Divide(F_t , C); // kernel
            D_1 = Merge(d,C, L_j );// co-kernel

            D = D ∪ D_1 ∪ F_1;
           D = D ∪ Kernels(j ,F_1 , D_1);
        }
     }
   }
 return D;
}

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

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

Merge(C_1, C_2, C_3)
{
    C_1, C_2, C_3 = cubes ;
    Cube M;
    for(i = 0; i < Number of literals; i++)
       M[i] = C_1[i] + C_2[i] + C_3[i];
   return M;
}
```

Figure 2. Algorithms for kernel and co-kernel extraction

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 i to |L| (the total number of literals) in the "for" loop are used to prevent the same kernels from being generated again.

As an example consider the expression P_1 in Figure 1a. The literal set for this example is L = {x,y,z}. Dividing first by x gives $F_t = x^2y + xy^2z$. Now the biggest cube dividing F_t is C = xy. Dividing F_t by xy gives the first kernel $F_1 = x + yz$ with the corresponding co-kernel $D_1 = x^2y$. Since F_1 does not have any kernels, F is now divided with y to obtain $F_t = x^3 + x^2yz$. $C = x^2$, but since x is a literal already encountered, the division is not performed as it would give the same kernel obtained before. The algorithm terminates here and finally we have two kernels, $K_1 = x + yz$ with co-kernel x^2y, and $K_2 = x^3y + x^2y^2z$ with co-kernel 1.

3.3 Extracting kernel intersections

Finding every intersection between the kernels is equivalent to finding all multiple cube common divisors. The task of finding the best kernel intersections can be modeled as a rectangle covering problem [10]. A matrix called the **kernel cube matrix** (KCM) is formed from the kernels and the co-kernels that have been generated. This matrix has one row for each kernel (co-kernel) that has been extracted and one column for each distinct cube of a kernel. An element (i,j) of the matrix is 1 if the kernel corresponding to row i contains the cube represented by column j, else it is a 0. A term may be covered by more than one element since the extracted kernels may overlap. Figure 3 is an example of a KCM for the set of expressions in Figure 1 a. The rows of the KCM are marked with the co-kernel of the kernel corresponding to the row.

		1	2	3	4	5
		x	yz	4	-yz	-x
1	4	$1_{(3)}$	$1_{(4)}$	0	0	0
2	x^2y	$1_{(1)}$	$1_{(2)}$	0	0	0
3	x	0	0	$1_{(3)}$	$1_{(5)}$	0
4	yz	0	0	$1_{(4)}$	0	$1_{(5)}$
5	xy	0	0	$1_{(6)}$	0	$1_{(7)}$

Figure 3. Example Kernel Cube Matrix

A kernel intersection appears in the matrix as a **rectangle**. A rectangle is a set of columns and a set of rows of the kernel cube matrix such that all elements of the rectangle are 1. The task of finding the best set of kernel intersections is equivalent to finding the best set of rectangles in the matrix, such that all the 1s in the matrix are covered. Since this problem is known to be NP complete [11], we employ a greedy strategy to pick the best valued prime rectangle in each iteration. A **prime rectangle** is a rectangle that is not covered by any other rectangle. The value function for selecting the best rectangle depends on the optimization target. In this work we optimize for the total energy consumption in computing the polynomial.

3.3.a Modeling value function for energy savings

Given a prime rectangle define the following parameters.

R = number of rows ; ***C*** = number of columns

M(R_i) = number of multiplications in row i (co-kernel) i.

M(C_i) = number of multiplications in column i (kernel-cube i)

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 $M(R_i) + M(C_i) + 1$ multiplications. The total number of multiplications represented by the whole rectangle = $\mathbf{R}*\sum_{\mathbf{C}}\mathbf{M}(\mathbf{C_i})+\mathbf{C}*\sum_{\mathbf{R}}\mathbf{M}(\mathbf{R_i})+\mathbf{R}*\mathbf{C}$

Each row in the rectangle has C – 1 additions for a total of *R*(C-1)* additions. By selecting the rectangle, we will have extracted a common factor with $\sum_{C} M(C_i)$ 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. Weighing the number of multiplications by 'm', where 'm' is the ratio of the energy consumption of a multiplier to an adder in the target library, the value function can be formulated as weighted sum of the reduction in the number of multiplications and additions by selecting the rectangle and is given by Equation I.

$$\mathbf{VF_l = m*\{(C-1)*(R+\sum_{R}M(R_i))+(R-1)*(\sum_{C}M(C_i)\}} \\ \mathbf{+(R-1)*(C-1)} \qquad \textbf{(I)}$$

For a given supply voltage and clock frequency, the total power consumption of a datapath depends on a number of factors such as the number and type of functional units, the switching capacitance in the multiplexers, control units and interconnects. Besides the power consumption also depends on the scheduling and binding of the operations on the functional units. Furthermore, there are a number of different transformations such as retiming and pipelining of the operations that affect the total power consumption in a datapath [8]. Developing an automated technique that considers all these factors and optimizes the total power consumption of a datapath is a very difficult problem and developing tools that give different implementation choices to the designers is of practical interest [8, 12].

The value function in equation I assumes that the major component of active power dissipation in a datapath is due to the switching power in the functional units, and reducing the number of energy intensive operations will reduce the total energy consumption. The value function assumes that there is only a single type of adder and a single type of multiplier in the design library.

If the resource constraints are such that only a single adder and a single multiplier can be used, the value function (Equation I) will also produce a design with shorter latency compared to the other known techniques, and the power consumption can be reduced by voltage scaling. In the case of multiple functional units, the factoring of polynomials performed by our technique can actually increase the critical path length of the polynomial and the amount of improvement over other techniques is case dependant. For some polynomials, CSE may produce a shorter critical path, but at the expense of greater switching activity per control cycle. Our value function assumes that the savings in the number of energy intensive operations will outweigh the benefits of reduced latency (if any) produced by other methods.

3.3.b Algorithm for selecting kernel intersections

The algorithm for finding the kernel intersections is shown in Figure 4 and it is analogous to the **Distill** procedure [9] in logic synthesis. It is a greedy algorithm in which the best rectangle is extracted and substituted in each iteration. In the outer loop, the set of kernels is computed for the expressions $\{P_i\}$ and the kernel cube matrix is formed from that. Each iteration in the inner loop selects the most favorable rectangle if present, based on our value function. This rectangle is added to the set of expressions and a new literal is introduced to represent this new rectangle. The kernel cube matrix is then updated by removing those 1's in the matrix that correspond to the terms covered by the selected rectangle.

As an example consider the same set of expressions in Figure 1a. The set of kernels and co-kernels for each expression are:
P_1: $[x + yz](x^2y)$, $[x^3y + x^2y^2z](1)$
P_2: $[4 – x](yz)$, $[4 – yz](x)$, $[x + yz](4)$, $[4x + 4yz + xy^2z](1)$.
P_3: $[4 - x](xy)$, $[4xy – x^2y](1)$.

```
FindKernelIntersections( {P_i}, {L_i})
{
    while(intersections present)
    {
      D = FindKernels({P_i}, {L_i});
      KCM = Form Kernel Cube Matrix (D)
      {R} =Set of new kernel intersections = φ ;
      {V} =Set of new variables = φ;
      while(favorable rectangles exist)
      {
        {R} = {R} ∪ Best Rectangle;
        {V} = {V} ∪ New Literal
        Update KCM;
      }
      Rewrite {P_i} using{R};
      {P_i} = {P_i} ∪ {R};
      {L_i} = {L_i} ∪ {V};
    }
}
```

Figure 4. The algorithm for extracting kernel intersections

The kernel cube matrix is formed as shown in Figure 3. The kernels which are the original expressions (with co-kernel '1') are not shown in the figure to simplify representation. The rows correspond to the five co-kernels generated and the columns correspond to the five distinct kernel cubes.

Consider the rectangle consisting of the rows {1,2}, and the columns {1,2}. This rectangle has R = 2 rows, C = 2 columns, total number of multiplications in the rows $\sum_{R} M(R_i) = 2$ and total number of multiplications in the columns $\sum_{C} M(C_i) = 1$. Using our value function (equation I) for estimating energy savings, and using a scaling factor m = 40 for the multiplier, the value of this rectangle can be computed to be 201 units. The value of this rectangle happens to be the best and is equal to the value of the rectangle consisting of rows {4,5} and {3,5}, but the first rectangle is chosen arbitrarily. Because the rectangle covers the terms 1,2,3 and 4, the elements in the matrix corresponding to these terms are set to 0. In the next iteration, the rectangle corresponding to row {5} and columns {3,5}, which has a value of 40 is selected. No more rectangles can be extracted in the subsequent

iterations. Now we have two literals d_1 and d_2 corresponding to the rectangles (x + yz) and (4 – x) respectively. The set of expressions can now be written as shown in Figure 5.

$P_1 = x^2yd_1$
$P_2 = 4d_1 - xyz$
$P_3 = xyd_2$
$d_1 = x + yz$
$d_2 = 4 - x$

Figure 5. Expressions after extracting kernel intersections

3.4 Extracting cube intersections

The procedure for finding kernel intersections finds the best multiple cube common factors. We can further optimize the result by finding single cube common factors among the set of cubes. A matrix called the **cube literal incidence matrix** is formed where there is a row for each cube in the set of expressions and a column for each literal. An element (i,j) of this matrix has a value equal to the power of the variable j in the cube i. The cube intersections appear as a rectangle in the matrix. A **rectangle** for this matrix is defined as a set of rows and columns in the matrix such that all elements are non-zero. The common cube C thus extracted will have the variable powers assigned to the minimum powers of the corresponding variables (columns) in the rectangle. The value of selecting the cube intersection can be computed from the resulting savings in the number of multiplications. Let $\sum C_i$ be the sum of the 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 function for selecting a rectangle with R rows is given by the equation

$$\mathbf{VF_2 = m * (R-1)*(\sum C_i - 1)} \qquad \mathbf{(II)}$$

Term	+/-	x	y	z	4	d_1	d_2
1	+	2	1	0	0	1	0
2	+	0	0	0	1	1	0
3	-	1	1	1	0	0	0
4	+	1	1	0	0	0	1
5	+	1	0	0	0	0	0
6	+	0	1	1	0	0	0
7	+	0	0	0	1	0	0
8	-	1	0	0	0	0	0

Figure 6. Cube Literal Incidence matrix

The algorithm for extracting cube intersections is similar to that of extracting kernel intersections (Figure 3), except that the rectangles now correspond to cube intersections. As an example, consider the set of arithmetic expressions in Figure 5 obtained after finding kernel intersections. The cube literal incidence matrix for these expressions is shown in Figure 6. From the example, it can be observed that the rectangle corresponding to the rows {1,3,4} and columns corresponding to the variable {x,y} which represents the cube C = xy is the most favorable rectangle since it saves two multiplications. No more cube intersections are detected in the subsequent iterations. A literal $d_3 = xy$ is added to the list of literals and the final set of expressions can be written as shown in Figure 1d.

4. Experimental results

We investigated a few polynomials commonly found in multivariate interpolation schemes and DSP functions [13, 14]. We compared the number of multiplications and additions/subtractions obtained by our technique with those in the original polynomials and also with those obtained by applying common subexpression elimination (CSE) and Horner transform (Table 1). The first four examples (**ex1 to ex4**) are polynomials used for multivariate polynomial interpolation, and are useful in Computer Graphics applications for modeling free form curves and surfaces. The examples ex1 to ex4 are respectively **quartic-spline, quintic-spline, chebyshev and cosine-wavelet.** The fifth example (**ex5**) is a set of polynomials from **FIR filter** [14], where the trigonometric functions Sine and Cosine have been approximated by their Taylor series expansions.

We used an energy scaling factor m = 15 for our experiments (Equations I and II), based on the average ratio of energy consumption for the adder and multiplier in our technology library. From Table 1, we can observe that our technique gives the least number of multiplications compared to all the other techniques. But the number of additions remains the same in all the cases, since there are no multiple term common subexpressions in these examples. The reduction in the number of operations is because of factorization and elimination of single cube (term) common subexpressions, which reduces the number of multiplications.

Table 1. Comparing number of multiplications(M) and additions/subtractions(A)

	Original		**CSE**		**Horner**		**Our Technique**	
	M	**A**	**M**	**A**	**M**	**A**	**M**	**A**
ex1	23	4	16	4	17	4	13	4
ex2	34	5	22	5	23	5	16	5
ex3	32	8	18	8	18	8	11	8
ex4	43	17	24	17	19	17	17	17
ex5	34	6	23	6	20	6	13	6
Avg	**33.2**	**8**	**20.6**	**8**	**19.4**	**8**	**14.0**	**8**

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 5V. 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 2 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 29.3% over CSE and

26.1% over Horner), and energy delay product (average 45.4% over CSE and 43% over Horner).

Table 2. Improvement over CSE and Horner with minimum hardware constraints

	Area (%) (I)		Energy (5 V) (%) (II)		Energy-Delay (%) (III)		Energy (scaled V) (%) (IV)	
	C	H	C	H	C	H	C	H
ex1	7.5	0.1	13.6	25.6	20.4	39.4	24.6	49.5
ex2	0.3	-4.2	21.6	29.3	39.0	48.8	52.2	64.6
ex3	-7.5	-24.2	29.4	10.4	47.6	25.9	62.2	36.9
ex4	5.6	2.5	37.0	28.7	57.1	46.1	74.3	59.8
ex5	3.7	2.0	44.8	36.8	62.8	54.8	78.3	69.7
Avg	**1.9**	**-4.8**	**29.3**	**26.1**	**45.4**	**43.0**	**58.3**	**56.1**

Since 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 58.3% over CSE(C) and 56.1% reduction over Horner (H) with voltage scaling.

Table 3a. Improvement over CSE (C) and Horner (H) with medium hardware constraints

	Area (%)		Energy (5 V) (%)		Energy-Delay (%)		Energy (scaled V) (%)	
	C	H	C	H	C	H	C	H
ex1	30.5	3.9	16.1	39.2	9.7	44.1	9.7	55.0
ex2	14.8	1.0	9.7	29.6	20.3	58.7	22.7	75.4
ex3	8.3	3.7	42.5	29.1	44.9	37.0	51.8	45.0
ex4	8.9	9.0	28.2	29.5	39.5	40.6	47.4	48.3
ex5	8.0	6.6	41.4	40.8	58.4	59.7	72.6	75.9
Avg	**14.1**	**4.9**	**27.6**	**33.6**	**34.6**	**48.0**	**40.8**	**60.0**

Table 3b. Multipliers (M) and adders (A) allocated with medium hardware constraints

	CSE		Horner		Our Technique	
	M	A	M	A	M	A
ex1	3	1	2	1	2	1
ex2	4	1	4	1	4	1
ex3	3	1	3	2	3	2
ex4	4	1	4	2	4	2
ex5	3	1	3	1	3	2

Table 3a shows the synthesis results for medium hardware constraints. Medium hardware implementations trade off area for energy efficiency. Though they have 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 3b 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. The results show a significant reduction in total energy consumption (average 27.6% over CSE(C) and 33.6% over Horner (H) scheme), as well as energy delay product (average 34.6% over CSE and 48.0 % over Horner). The delay for Horner scheme is much worse than both CSE and our technique, because the nested additions and multiplications of Horner scheme typically results in a longer critical path. Using voltage scaling, we estimated an average energy reduction of 40.8% over CSE and 60 % over Horner scheme.

5. Conclusions

In this work, we presented a new technique to optimize the energy consumption for computing polynomial expressions, by reducing the number of power hungry operations given a fixed supply voltage. Our technique has shown significant reduction in the energy consumptions for custom hardware computations of polynomial expressions, compared to the conventional common subexpression elimination and Horner transform. The average run time of our algorithm is only 0.42s for the set of benchmark polynomial examples. Our technique can be integrated into conventional high level synthesis framework for optimizing applications with polynomial expressions. In this work we have assumed a library with only one kind of adder and multiplier, and a single supply voltage. In future we would like to develop techniques that can take advantage of different types of functional units, operating at multiple voltages.

References

1. A.Peymandoust and G.D. Micheli. *Using Symbolic Algebra in Algorithmic Level DSP Synthesis*. in *Design Automation Conference*. 2001.
2. V.Krishna, N.Ranganathan, and N.Vijayakrishnan. *An Energy Efficient Scheduling Scheme for Signal Processing Applications*. in *VLSI Design*. 1999.
3. A.Peymandoust, T.Simunic, and G.D. Micheli, *Low Power Embedded Software Optimization Using Symbolic Algebra*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 2002.
4. R.H.Bartels, J.C.Beatty, and B.A.Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. 1987: Morgan Kaufmann Publishers, Inc.
5. *GNU C Library*.
6. S.S.Muchnick, *Advanced Compiler Design and Implementation*. 1997: Morgan Kaufmann Publishers.
7. *Maple*. 1998, Waterloo Maple Inc.
8. A.P.Chandrakasan, et al., *Optimizing power using transformations.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 1995. **14**(1): p. 12-31.
9. R.K.Brayton and C.T.McMullen. *The Decomposition and Factorization of Boolean Expressions*. in *International Symposium on Circuits and Systems*. 1982.
10. R.Brayton, R.R., A. Sangiovanni Vincentelli and A.Wang. *Multi-level Logic Optimization and the Rectangular Covering Problem.* in *International Conference on Compute Aided Design*. 1987.
11. R.Rudell, *Logic Synthesis for VLSI Design*. 1989, PhD Thesis, University of California, Berkeley.
12. Macii, E., M. Pedram, and F. Somenzi, *High-level power modeling, estimation, and optimization.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 1998. **17**(11): p. 1061-1079.
13. G.Nurnberger, J.W.Schmidt, and G.Walz, *Multivariate approximation and splines*. 1997: Springer Verlag.
14. P.M.Embree, *C Algorithms for Real-Time DSP*. 1995: Prentice Hall.