Optimizing Polynomial Expressions by Factoring and Eliminating Common Subexpressions

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 compute a wide variety of mathematical functions commonly found in signal processing and graphics applications, which provide good opportunities for optimization. However existing compiler techniques for reducing code complexity such as common subexpression elimination and value numbering are targeted towards general purpose applications and are unable to fully optimize these expressions. This paper presents algorithms to reduce the number of operations to compute a set of polynomial expression by factoring and eliminating common subexpressions. These algorithms are based on the algebraic techniques for multi-level logic synthesis. Experimental results on a set of benchmark applications with polynomial expressions showed an average of **42.5%** reduction in the number of multiplications and **39.6%** reduction in the number of clock cycles for computation of these expressions on the ARM processor core, compared to common subexpression elimination.

1. INTRODUCTION

The rapid advancement and specialization of embedded systems has pushed the compiler designers to develop application specific techniques to get the maximum benefit at the earliest stages of the design process. There are a number of embedded system applications that have to frequently compute polynomial expressions[1, 2]. Polynomial expressions are present in a wide variety of applications domains since any continuous function can be approximated by a polynomial to the desired degree of accuracy [3, 4]. There are a number of scientific applications that use polynomial interpolation on data of physical phenomenon. These polynomials need to be computed frequently during simulation. Adaptive signal processing applications use polynomials to describe a wide variety of filtering operations [5, 6]. Real time computer graphics applications often use polynomial interpolations for surface and curve generation, texture mapping etc. [7]. Most DSP algorithms often use trigonometric functions like Sine and Cosine which can be approximated by their Taylor series as long as the resulting inaccuracy can be tolerated [1, 2].

There are a number of compiler optimizations for reducing code complexity such as value numbering, common subexpression elimination, copy propagation, strength reduction etc. These optimizations are performed both locally in a basic block and globally across the whole procedure, and are typically applied at a low level intermediate representation of the program [8]. But these optimizations have been developed for general purpose applications and are unable to fully optimize polynomial expressions. Most importantly, they are unable to perform factorization, which is a very effective technique in reducing the number of multiplications in a polynomial expression. For example, consider the evaluation of the trigonometric identity sin(x) in Figure 1a, which has been approximated using Taylor series. The subscripts under each term denote the term numbers in the expression.

$$\sin(x) = x_{(1)} - S_3x^3_{(2)} + S_5x^5_{(3)} - S_7x^7_{(4)}$$

$$S_3 = 1/3!\ ,\ S_5 = 1/5!,\ S_7 = 1/7!$$

Figure 1a. Evaluation of sin(x)

A naïve implementation of this expression will result in 15 multiplications and three additions/subtractions. Common subexpression elimination (CSE) is typically applied on a low level intermediate program representation, and it iteratively detects and eliminates two operand common subexpressions [8]. When CSE is applied to the expression in Figure 1a, the subexpression $d_1 = x*x$ is detected in the first iteration. In the second iteration, the subexpression $d_2 = d_1*d_1$ is detected, and the algorithm stops. The optimized expression is now written as:

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

Figure 1b. Using common subexpression elimination

These expressions consist of a total of nine multiplications and three additions/subtractions, giving us a saving of six multiplications over the previous evaluation. Using our algebraic techniques that we develop in this paper, we obtain the set of expressions shown in Figure 1c. These

expressions now consist of a total of five multiplications and three additions/subtractions. Therefore there is a saving of four multiplications compared to the previous technique and a saving of 10 multiplications compared to the original representation.

It should be noted that this representation is similar to the hand optimized form of the Horner transform used to evaluate trigonometric functions in many libraries [9] including the GNU C library [10].

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

Figure 1c. Using algebraic techniques

We were able to perform this optimization by factorizing the expression and finding common subexpressions. It is impossible to perform such optimizations using the conventional compiler techniques. We show that such optimizations can be done automatically for any set of arithmetic expressions using algebraic techniques.

Decomposition and factoring are the major techniques in multi-level logic synthesis, and are used to reduce the number of literals in a set of Boolean expressions [11, 12]. In this paper, we show how the same concepts can be used to reduce the number of operations in a set of arithmetic expressions.

The main contributions of our paper are:

- Transformation of the set of arithmetic expressions that allow us to detect all possible common subexpressions and factors.
- Algorithms to find the best set of factors and common subexpressions to reduce number of operations.
- Experimental results on a set of benchmark applications where we show the superiority of our technique over CSE and Horner transform.

The rest of the paper is organized as follows. Section 2 presents the transformation of polynomial expressions. Section 3 presents algorithms to optimize the transformed polynomials. We present some related work on arithmetic expressions in Section 4. Experimental results are presented in Section 5. We conclude and talk about future work in Section 6.

2. 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 subexpressions and writing them in matrix form.

2.1 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 non-negative 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 sin(x) is represented as shown in Figure 2.

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

Figure 2. Matrix representation of polynomial expressions

2.2 Definitions

We use the following terminology 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 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 ($+3a^2b + (-2a^3b^2c)$ + …). An SOP expression is said to be **cube-free** if there is no 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. A kernel should have 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.

2.3 Generation of kernels and co-kernels

The importance of kernels is illustrated by the following theorem [11].

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 kernel intersections we can find all multiple term common subexpressions. The product of the kernel and its corresponding co-kernel gives back the original terms that the kernel covers. Therefore a kernel and its corresponding co-kernel provide a possible factorization opportunity. Since the set of kernels is much

smaller than the set of all divisors, finding intersections amongst kernels leads to an efficient algorithm.

The algorithm for extracting all kernels and co-kernels of a set of polynomial expressions is shown in Figure 3. This algorithm is analogous to kernel generation algorithms in [11], and has been generalized to handle polynomial expressions. The main difference is the way division is performed in our technique, where the values of the elements in the divisor are subtracted from the suitable set of rows in the dividend. Besides, several key steps in the algorithm have been modified to handle polynomial expressions of any order.

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 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] ≥ 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.

As an example, consider the expression F = sin(x) in Figure 1a. 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 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 are
$[x](1 - S_3x^2 + S_5x^4 - S_7x^6)$; $[x^3](-S_3 + S_5x^2 - S_7x^4)$;
$[x^5](S_5 - S_7x^2)$; $[1](x - S_3x^3 + S_5x^5 - S_7x^7)$;

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

2.4 Constructing Kernel Cube Matrix (KCM)

All kernel intersections and multiple term factors can be identified by arranging the kernels and co-kernels in a 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 Figure 5a. 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 parenthesis in each element represents the term number that it represents. A **rectangle** is a set of rows and set of columns of the KCM such that all the elements are '1'. The **value** of a rectangle is 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 weighted rectangular covering problem described in [13], which is NP-hard. We use a greedy iterative algorithm described in Section 3.1 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 following parameters, we can calculate its value:

R = number of rows ; C = number of columns

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

$M(C_i)$ = number of multiplications in column (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 is equal to $R*\sum_C M(C_i) + C*\sum_R M(R_i) + R*C$. 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 $\sum_C M(C_i)$ 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.

Therefore the value of a rectangle, which represents the savings in the number of operators by selecting the rectangle can be written as

$$\text{Value}_1 = \{(C-1)*(R+\sum_R M(R_i)) + (R-1)*(\sum_C M(C_i)\} + (R-1)*(C-1) \quad \text{(I)}$$

2.5 Constructing Cube Literal Incidence Matrix (CIM)

The 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 Figure 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 non-zero. 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 3.2, 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 the 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 the equation:

$$\text{Value}_2 = (R-1)*(\sum C[i] - 1) \quad \text{(II)}$$

3. ALGORITHMS FOR OPTIMIZATION

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.

3.1 Extracting kernel intersections

The algorithm for finding kernel intersections is shown in figure 4, and is analogous to the **Distill** procedure [11] 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 the 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 Equation I. 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.

```
FindKernelIntersections( {P_i}, {L_i})
{
    while(1)
    {
       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;
       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_i} = {L_i} ∪ {V};
    }
}
```

Figure 4. Algorithm for finding kernel intersections

For example, in the KCM for our sin(x) example shown in Figure 5a, consider the rectangle corresponding to the row {2} and columns {5, 6, 7}.This is a prime rectangle having R = 1 row and C = 3 columns, total number of multiplications in the rows $\sum_{R} M(R_i) = 2$ and total number of multiplications in the columns $\sum_{C} M(C_i) = 6$. Using our value function in Equation (I), the total number of operations (here multiplications) saved is 6.

		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								$1_{(3)}$	$1_{(4)}$

Figure 5a. Kernel cube matrix (1st iteration)

This is the most valuable rectangle, and is selected. Since this rectangle covers the terms 2, 3 and 4, the elements in the matrix corresponding to these terms are deleted. No more favorable rectangles are extracted in this KCM, and now we have two expressions, after rewriting the original expression for sin(x).

$\sin(x) = x_{(1)} + (x^3 d_1)_{(2)}$

$d_1 = (-S_3)_{(3)} + (S_5x^2)_{(4)} - (S_7x^4)_{(5)}$

Extracting kernels and co-kernels as before, we have the following KCM

		1	2	3	4
		1	x^2d_1	S_5	$-S_7x^2$
1	x	$1_{(1)}$	$1_{(2)}$		
2	x^2			$1_{(4)}$	$1_{(5)}$

Figure 5b. Kernel Cube matrix (2nd iteration)

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 2 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 Figure 5c.

$\sin(x) = (xd_3)_{(1)}$

$d_3 = (1)_{(2)} + (x^2d_1)_{(3)}$

$d_2 = (S_5)_{(4)} - (x^2S_7)_{(5)}$

$d_1 = (x^2d_2)_{(6)} - (S_3)_{(7)}$

Figure 5c. Expressions after kernel extraction

3.2 Extracting Cube Intersections

The procedure for finding cube intersections is used to find single term common subexpressions and is performed after finding kernel intersections. The algorithm for finding cube intersections is analogous to the **Condense** algorithm [11] in logic synthesis, and is shown in Figure 6. In each iteration of the inner loop, the best prime rectangle is extracted using the value function in Equation (II). 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).

```
FindCubeIntersections( {P_i}, {L_i})
{
    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};
       {L_i} = {L_i} ∪ {V};
    }
}
Collapse(M,B)
{
    ∀ rows i of M that contain cube B
       ∀ columns j of B
       M[i][j] = M[i][j] – B[j];
}
```

Figure 6. The algorithm for finding Cube intersections

As an example, consider the CIM in Figure 7 for the set of expressions in Figure 5c. 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 Equation (II), 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.

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

Figure 7. Cube literal incidence matrix example

A literal $d_4 = x^2$ is added to the expressions which can now be written as in Figure 1c.

4 RELATEDWORK

Common subexpression elimination (CSE) and value numbering are the conventional techniques for redundancy elimination in general purpose programs. Both techniques are applied on a low level intermediate program representation [8], where each expression contains a maximum of two operands. The amount of improvement depends on this intermediate representation which in turn depends upon the original expressions. For example if there are two expressions $F_1 = a*b*b$ and $F_2 = c*b*b$, then the common subexpression b*b will be detected only if the expressions are parenthesized as $F_1 = a*(b*b)$ and $F_2 = c*(b*b)$. Another drawback of these methods is that they are unable to perform factorization of polynomial expressions, which is an effective technique to reduce the number of multiplications in an expression. These techniques are sufficient for general purpose programs but are unable to fully optimize applications with polynomial expressions. Unfortunately there has not been enough work done to optimize these expressions.

Code generation for simple arithmetic expressions by efficient use of common subexpression elimination and register allocation has been well studied [14, 15]. In [16] a technique for factorizing arithmetic expressions was proposed. But this technique can factor at a time, expressions consisting of either only multiplications or additions, and hence cannot do the same optimizations as our technique.

Horner form is the default way of evaluating trigonometric functions in many libraries. 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 this method optimizes only one expression at a time and does not look for common subexpressions among a set of polynomial expressions. Besides it does not give good results for multivariate polynomials.

There has been some work on optimizing single variable polynomial expressions by exploiting the common integer powers of the variable [17] by viewing the integer powers as bit patterns. But this technique suffers from two major disadvantages. It can optimize only univariate polynomials and is unable to perform factorization of polynomial expressions.

Symbolic algebra has been shown to be a good technique for manipulating polynomials and it has been shown to have applications in both high level synthesis [1] and low power embedded software optimization [2]. Symbolic algebra techniques can extract non-trivial factors like $a^2 - b^2 = (a+b)(a-b)$ which our algebraic methods cannot. This is because our technique depends upon repeated occurrence of the same set of literals in multiple cubes to enable the extraction of common factors. But a major drawback of the symbolic algebra method is that its results depend heavily on a set of library expressions. In the above example, it will be able to factorize the expression to (a+b)(a-b) only if it has a library expression (a+b) or (a-b). It should be noted that this is different from having an adder in the library. In comparison, our technique does not depend on any library expressions. Besides, symbolic algebra techniques are good at optimizing only one polynomial expression at a time. But our technique can optimize any number of polynomial expressions together. Also our technique should be much faster than the exhaustive search on all library elements performed by their methods. Both [1] and [2] do not report the CPU run times of their algorithm.

5. EXPERIMENTAL RESULTS

The goal of our experiments was to investigate the usefulness of our technique in reducing the number of arithmetic operations in real life applications having a number of polynomial expressions. We found polynomial expressions to be prevalent in multimedia and signal processing applications [18, 19]and computer graphics applications [7]. In the examples 1 to 6, the polynomials were obtained from the basic blocks of the functions shown in the table. The application source code [18, 19] was compiled to Machine SUIF [20] intermediate representation from which we obtained the basic blocks. Our algorithm extracted arithmetic subgraphs from these basic blocks and optimized them using our techniques described in sections 2 and 3. To get bigger basic blocks and to optimize multiple polynomials we unrolled the loops in these examples. These functions consisted of a

	Application	Function	Unoptimized		Using CSE		Using Horner		Using our Algebraic Technique		Improvement over			
											CSE		Horner	
			A	M	A	M	A	M	A	M	M (%)	Clock cyles on ARM7 (%)	M (%)	Clock cycles on ARM7 (%)
1	MP3 decoder	hwin_init	80	260	72	162	80	110	64	86	46.9	44.0	21.8	21.6
2	MP3 decoder	imdct	63	189	63	108	63	90	63	54	50.0	44.7	40.0	35.1
3	Mesa	gl_rotation	10	92	10	34	10	37	10	15	55.9	52.8	59.5	56.4
4	Adaptive filter	lms	35	130	35	85	35	55	35	40	52.9	48.9	27.3	24.2
5	Gaussian noise filter	FIR	36	224	36	143	36	89	36	63	55.9	53.3	29.2	27.0
6	Fast convolution	FFT	45	194	45	112	45	83	45	56	50.0	46.3	32.5	29.3
7	Graphics	quartic-spline	4	23	4	17	4	20	4	14	17.6	16.8	30.0	28.8
8	Graphics	quintic-spline	5	34	5	22	5	23	5	16	27.3	26.1	30.4	29.2
9	Graphics	chebyshev	8	32	8	18	8	18	8	11	38.9	35.7	38.9	35.7
10	Graphics	cosine-wavelet	17	43	17	24	17	19	15	17	29.2	27.0	10.5	10.7
	Average		**30.3**	**122**	**29.5**	**72.5**	**30.3**	**54.4**	**28.5**	**37.2**	**42.5%**	**39.6%**	**32%**	**29.8%**

Table 1. Experimental results (comparing number of additions (A) and multiplications (M) using different methods)

number of trigonometric functions like Sine and Cosine which were approximated by their Taylor series expansions. The polynomials in examples 7 to 10 are polynomial models used for multivariate polynomial interpolation schemes in computer graphics [7]. The average run time for our algorithms was 0.45s.

We compared our technique with both Common subexpression elimination (CSE) and the Horner transformation. Local CSE [8] was applied to the basic blocks of the functions in the examples 1 to 6. For examples 7 to 10, we applied local CSE on an intermediate program representation of the polynomial expressions. Since the results of CSE depend upon the way the expressions are written, we parenthesized these expressions to enable CSE to detect the maximum number of common subexpressions. The Horner forms of the polynomials were obtained from calls to Maple [21]. In Table 1, we compare the number of additions (A) and multiplications (M) obtained from our technique with those obtained from CSE and Horner. We obtained an average savings of 42.5 % in the number of multiplications over CSE and 32% savings over Horner. The average improvement in the number of additions was only 2.3% over CSE and 3.2% over Horner. This was because most of the improvement obtained from these examples was due to efficient factorization of the polynomial expressions and from finding single term common subexpressions, which only reduces the number of multiplications. We also estimated the savings in the number of clock cycles on the ARM7TDMI core [22] which is one of the industry's most popular 32-bit RISC microprocessor. It is a sequential processor with a three stage instruction pipeline. The ARM7TDMI features a multi-cycle 32x32 fixed-point multiplier that multiplies multiplicands in 8-bit stages, and has a worst case latency of five clock cycles. By weighing additions by one clock cycle and multiplications by five clock cycles, we observed an average 39.6% improvement in clock cycles compared to CSE and an average 29.8% savings in the number of clock cycles compared to Horner.

In summary, our results show that for a given set of polynomial expressions, our technique gives the best results in terms of savings in the number of operations. These optimizations are impossible to obtain using the conventional compiler techniques.

6. CONCLUSIONS

The key contribution of our work is the development of a new methodology that can factor and eliminate common subexpressions in a set of polynomial expressions of any order and containing any number of variables. These techniques have been adapted from the algebraic techniques that have been established for multi-level logic synthesis. Our technique is superior to the conventional

optimizations for polynomial expressions. Experimental results have demonstrated the benefits of our technique. We believe that these algorithms give significant opportunities for optimizing performance and power consumption in embedded applications

Right now our technique does not consider the effect of other compiler optimizations on the optimizations done by our technique. In future we would like to integrate our technique with the conventional compiler optimization pass and observe the actual improvement in total application run time.

REFERENCES

[1] A.Peymandoust and G. D. Micheli, "Using Symbolic Algebra in Algorithmic Level DSP Synthesis," *Proceedings of the Design Automation Conference*, 2001.

[2] 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.

[3] D.E.Knuth, *The Art of Computer Programming*, vol. 2: Seminumerical algorithms, Second ed: Addison-Wesley, 1981.

[4] C.Fike, *Computer Evaluation of Mathematical Functions*. Englewood Cliffs, N.J: Prentice-Hall, 1968.

[5] V.J.Mathews, "Adaptive Polynomial Filters," *IEEE Signal Processing Magazine*, vol. 8, pp. 10-26, 1991.

[6] J.L.Wu and Y.M.Huang, "Modularised Fast Polynomial Transform Algorithm for Two Dimensional Digital Signal Processing," *IEEE Proceedings on Radar and Signal Processing*, vol. 137, pp. 253-261, 1990.

[7] R.H.Bartels, J.C.Beatty, and B.A.Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*: Morgan Kaufmann Publishers, Inc., 1987.

[8] S.S.Muchnick, *Advanced Compiler Design and Implementation*: Morgan Kaufmann Publishers, 1997.

[9] R.Green, "Faster Math Functions," Sony Computer Entertainment Research and Development 2003.

[10]"GNU C Library."

[11]R.K.Brayton and C.T.McMullen, "The Decomposition and Factorization of Boolean Expressions," *Proceedings of the International Symposium on Circuits and Systems*, 1982.

[12]A.S.Vincentelli, A.Wang, R.K.Brayton, and R.Rudell, "MIS: Multiple Level Logic Optimization System," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1987.

[13]R.K.Brayton, R.Rudell, A.S.Vincentelli, and A.Wang, "Multi-level Logic Optimization and the Rectangular Covering Problem.," *Proceedings of the International Conference on Compute Aided Design*, 1987.

[14]A.V.Aho, S.C.Johnson, and J.D.Ullman, "Code Generation for Expressions with Common Subexpressions," *ACM*, vol. 24, pp. 146-160, 1977.

[15]R.Sethi and J.D.Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Communications of the ACM*, vol. 17, pp. 715-728, 1970.

[16]M.A.Breuer, "Generation of Optimal Code for Expressions via Factorization," *Communications of the ACM*, vol. 12, pp. 333-340, 1969.

[17]M.Potkonjak, M.B.Srivastava, and A.P.Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1996.

[18]C.Lee, M.Potkonjak, and W.H.Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," *Proceedings of the International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, United States, 1997.

[19]P.M.Embree, *C Algorithms for Real-Time DSP*: Prentice Hall, 1995.

[20]M.D.Smith and G.Holloway, "An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization," Harvard University, Cambridge, Mass. 2000.

[21]"Maple," Waterloo Maple Inc, 1998.

[22]"ARM7TDMI Technical Reference Manual," 2003.