Common Subexpression Elimination Involving Multiple Variables for Linear DSP Synthesis

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

Common subexpression elimination is commonly employed to reduce the number of operations in DSP algorithms after decomposing constant multiplications into shifts and additions. Conventional optimization techniques for finding common subexpressions can optimize constant multiplications with only a single variable at a time, and hence cannot fully optimize the computations with multiple variables found in matrix form of linear systems like DCT, DFT etc. In this paper we transform these computations such that all common subexpressions involving any number of variables can be detected. We then present heuristic algorithms to select the best set of common subexpressions. Experimental results show the superiority of our technique over conventional techniques for common subexpression elimination.

1. INTRODUCTION

The present generation embedded systems have stringent requirements on performance and power consumption. Many embedded systems employ DSP algorithms for communications, image processing, video processing etc, which are very compute intensive. Custom hardware implementation of these computation intensive DSP kernels is a good solution to meet the requirements for latency and power consumption. These DSP algorithms contain a large number of multiplications with constants [1]. Decomposing these constant multiplications into shifts and additions leads to an efficient hardware implementation. Finding common subexpressions in the set of additions further reduces the complexity of the implementation.

Conventional methods for finding common subexpressions rely on finding common digit patterns in the set of constants multiplied by a single variable. The common subexpressions correspond to the common partial products formed during the multiplication of the variable with the constants. Finding all possible common digit patterns can extract all possible common subexpressions when all the constant multiplications are with a single variable such as found in the transformed form of FIR digital filters [2, 3]. Many DSP transforms like Discrete Cosine Transform (DCT) and Discrete Fourier Transform (DFT) can be expressed as a multiplication of a constant matrix C with a vector of input samples X, where each output signal Y[i] is of the form shown in Equation I.

$$Y[i] = \sum_{j=0}^{N-1} C_{i,j} * X[j] \qquad \text{(I)}$$

Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)
1063-6862/04 $ 20.00 IEEE

These systems consist of constant multiplications with multiple variables. Therefore conventional methods that consider one variable at a time [2, 4-7] will not be able to do a good optimization.

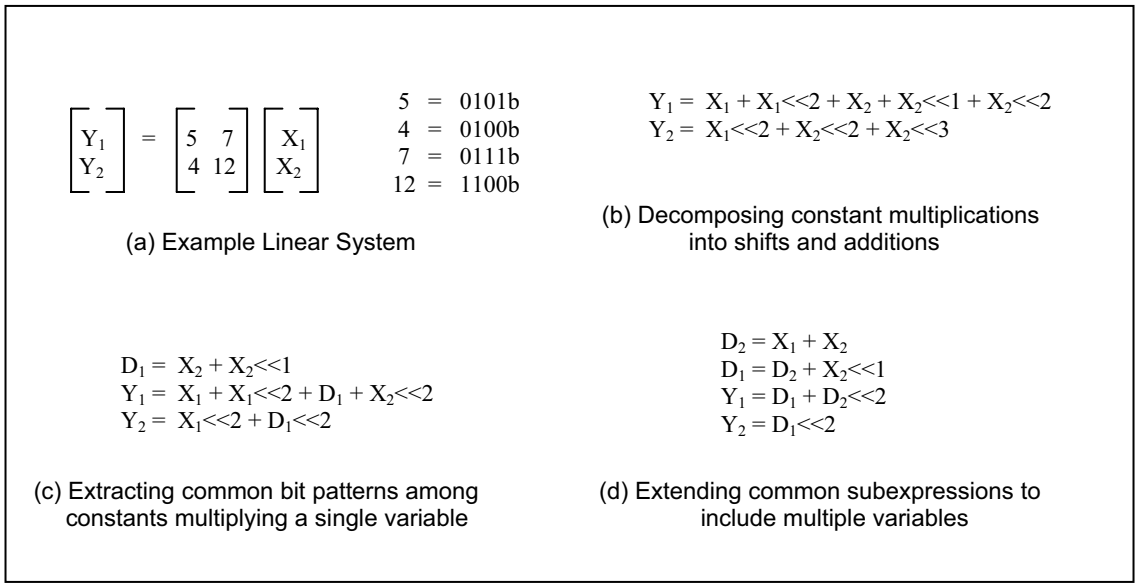

Figure 1. Example showing the improvement obtained by extending common subexpressions to include multiple variables

Consider the linear system shown in Figure 1a, which has two variables X_1 and X_2. Assume the binary representation for constants. The constants in column 1 (5 and 4) are multiplied with X_1 and the constants in column 2 (7 and 12) are multiplied with variable X_2. Decomposing the constant multiplications into shifts and additions we have an implementation with six shifts and six additions (Figure 1b). Finding common bit patterns in the constants multiplying the same variable, the pattern "11" is detected between the constants 7(0111) and 12 (1100), which multiply variable X_2. Extracting that common subexpression, we have an implementation with five shifts and five additions (Figure 1c). If the common subexpressions are extended to include multiple variables, we can have an implementation with just three additions and three shifts (Figure 1d). Therefore by extending common subexpressions to include multiple variables, the number of additions has reduced by 40% compared to methods which extract common subexpressions including only one variable. To the best of our knowledge there is no known technique that can find such common subexpressions for the matrix form of linear systems.

In this paper, we present a new polynomial transformation of the linear systems that enables us to detect common subexpressions with multiple variables in addition to those subexpressions involving a single variable. We present heuristic algorithms to extract the best set of common subexpressions for an implementation with a minimal number of additions/subtractions. The rest of the paper is organized as follows. We present some related work on the finding common subexpressions related to multiple constant multiplications in Section 2. We present our polynomial formulation of linear systems in Section 3. In Section 4, we present the heuristic algorithms to extract the common subexpressions. We present experimental results in Section 5. We finally conclude and talk about future work in Section 6.

2. RELATED WORK

There have been a number of papers on the problem of constant multiplication optimization for linear systems, particularly for digital filters [1-4, 6-8]. In this section we will present a brief summary of some of these works and how they compare with our techniques.

In [1], a scaling transformation of the constants followed by a bipartite matching algorithm was presented. Their algorithm cannot find all possible common digit patterns since it does not consider common patterns having different digit positions, which can be obtained by looking at different shifted forms of the constants. The authors in [1] do suggest a post processing step for the matrix form of linear systems where the expressions after the first optimization step (the y[i]'s in Equation I) are viewed as bit patterns, where each non-zero bit represents a unique subexpression obtained from the first optimization stage. Finding and eliminating common bit patterns among these sets of patterns, which now consist of multiple variables further reduces the number of additions/subtractions. But since this work does not consider the elimination of multiple variable common subexpressions together with single variable common subexpressions it cannot give good results, as our experimental results show. In [2], an exhaustive bit pattern enumeration algorithm was employed to extract all common digit patterns multiplying a single variable. The work in [4] targeted matrix form of linear systems, where a matrix splitting technique followed by an algorithm to find common subexpressions among the constants multiplying each variable was presented. In all the works mentioned above, the constant multiplication optimization was either performed for only for a single variable, or was extended to multiple variables as a post processing step. Our method provides a framework where all common subexpressions involving any number of variables can be detected concurrently, which leads to better results.

The work in [8] was the first attempt to extend common subexpression elimination to multiple variables, and was applied to the direct form of FIR filters. But the presented algorithms could only find 2-bit common subexpressions. Furthermore, they could not find common subexpressions across shifted forms of the constants. In [3], it was shown how using different representations for the constants can lead to better results compared to using a single representation like Canonical Signed Digit (CSD). But the exponential number of possible representations for the constants does not lead to a practical solution for a problem which by itself has an exponential complexity. In this paper we assume that the representations for the constants such as two's complement or CSD are given.

3. TRANSFORMATION OF CONSTANT MULTIPLICATIONS

In this section, we first introduce the polynomial representation of the constant multiplications. We then show how a subset of all algebraic divisors of the polynomial expressions (kernels) can detect all possible common subexpressions in a linear system.

3.a Polynomial representation

Using a given representation of the integer constant C, the multiplication with the variable X can be represented as

$$\mathbf{C*X} = \sum_{i} \pm (XL^{i})$$

where L represents the left shift from the least significant digit and the i's represent the digit positions of the non-zero digits of the constant, 0 being the digit position of the least significant digit. Each term in the polynomial can be positive or negative depending on the sign of the non-

zero digit. Thus this representation can handle signed digit representations such as the Canonical Signed Digit (CSD).

For example the constant multiplication $(12)_{decimal}$ *X = $(1100)_{binary}$*X = $XL^2 + XL^3$ in our polynomial representation. In case of systems where the constants are real numbers represented in fixed point, the constant can be converted into an integer, and the final result can be corrected by shifting right. For example in the constant multiplication $(0.101)_{binary}$ * X = $(101)_{binary}$*X*2^{-3} = $(X + XL^2)$*2^{-3} in our polynomial representation. The linear system in Figure 1a and 1b can be written using our polynomial formulation as shown in Figure 2. The subscripts in parenthesis represent the term numbers which will be used in our optimization algorithm.

$$Y_1 = {}_{(1)}X_1 + {}_{(2)}X_1L^2 + {}_{(3)}X_2 + {}_{(4)}X_2L + {}_{(5)}X_2L^2$$

$$Y_2 = {}_{(6)}X_1L^2 + {}_{(7)}X_2L^2 + {}_{(8)}X_2L^3$$

Figure 2. Polynomial representation of example linear system.

3.b Generating kernels of polynomial expressions

A **kernel** of a polynomial expression is a subexpression that is derived from the original expression through division by an exponent of L, and contains at least one term with a zero exponent of L, and all the terms of the original expression that had higher exponents of L. A **divisor** of a polynomial expression is a subexpression of the expression having at least one term with a zero exponent of L. The set of kernels of a polynomial expression is a subset of the set of divisors of the expression. The algorithm for generating kernels of a set of polynomial expressions is shown in Figure 3. The algorithm recursively finds kernels within kernels generated by dividing by the smallest exponent of L. The exponent of L that is used to divide the original expression to obtain the kernel is the corresponding **co-kernel** of the kernel expression.

```
Kernels (Pi)
{
   {D} = set of kernels and co-kernels = φ ;
    L = 1;

   if(Pi is a kernel)
     {D} = {D} ∪ (Pi, 1);

    while
   (at least 2 terms with non-zero  exponents of L exist in Pi)
   {
      MinL = Minimum (non-zero) exponent of L;
       Pi     = Divide(Pi , MinL);
       L      =  L * MinL;
      {D}   =  {D} ∪ (Pi, L);
    }
}
Divide(Pi, MinL)
{
  {Ti} = Terms of Pi having non-zero exponent of  L;
  Divide the exponent of L in each term in {Ti} by MinL;
   return {Ti};
 }
```

Figure 3. Algorithm for generation of kernels

For example consider the expression Y_1 in Figure 2. Since Y_1 satisfies the definition of a kernel, we record Y_1 as a kernel with co-kernel '1'. The minimum non-zero exponent of L in Y_1 is L. Dividing by L we obtain the kernel expression $X_1L + X_2 + X_2L$, which we record as a kernel with co-kernel L. This expression has L as the minimum non-zero exponent of L. Dividing by L, we obtain the kernel $X_1 + X_2$ with co-kernel L^2. No more kernels are generated and the algorithm terminates. For expression Y_2 we obtain the kernel $X_1 + X_2 + X_2L$ with co-kernel L^2.
The set of kernels and co-kernels for the expressions in Figure 2 are shown in Figure 4.

$({}_{(1)}X_1 + {}_{(2)}X_1L^2 + {}_{(3)}X_2 + {}_{(4)}X_2L + {}_{(5)}X_2L^2)[1]$
$({}_{(2)}X_1L + {}_{(4)}X_2 + {}_{(5)}X_2L)[L]$
$({}_{(2)}X_1 + {}_{(5)}X_2)[L^2]$
$({}_{(6)}X_1 + {}_{(7)}X_2 + {}_{(8)}X_2L)[L^2]$

Figure 4. Set of kernels and co-kernels for expressions in Figure 2.

The importance of kernels is illustrated by the following theorem.

Theorem: There exists a k-term common subexpression if and only if there is a k-term non-overlapping intersection between at least 2 kernels.

This theorem basically states that each common subexpression in the set of expressions appears as a non-overlapping intersection among the set of kernels of the polynomial expressions. Here a non-overlapping intersection implies that the set of terms involved in the intersection are all distinct. As an example of overlapping terms, consider the binary constant "1001001" represented in Figure 5. Converting the multiplication of this constant with the variable X into our polynomial representation, we get the expression ${}_{(1)}X + {}_{(2)}XL^3 + {}_{(3)}XL^6$. There are two kernels generated: $X + XL^3 + XL^6$ covering terms 1,2 and 3 and $X + XL^3$, covering terms 2 and 3. An intersection between these two kernels will detect two instances of the subexpression $X + XL^3$, but they overlap as they both cover term 2. This overlap can be seen in the overlap between the two instances of the bit-pattern "1001" as seen in Figure 5.
Implementing the polynomial $X + XL^3 + XL^6$ with two instances of $X + XL^3$ will result in the second term XL^3 being covered twice. Though we can still implement it this way, and subtract XL^3 at the end, it will not be beneficial in this case, as it would result in more additions than the original polynomial representation.

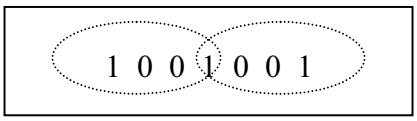

Figure 5. Overlap between kernel expressions

Proof:
If: If there is a k-term non-overlapping intersection among the set of kernels, then it means there is a multiple occurrence of the k-term subexpression in the set of polynomial expressions, and the terms involved in the intersection are all distinct. Therefore a k-term non-overlapping intersection in the set of kernels implies a k-term common subexpression.
Only If: Assume there are two instances of the k-term common subexpression. First assume the common subexpression satisfies the definition of a divisor, which means there is at least one term with a non-zero exponent of L. Now each instance of the subexpression will be a part of some kernel expression. This is because from the kernel generation algorithm (Figure 3), each kernel is generated recursively dividing through the lowest (non-zero) exponent of L. Each time a division by L is performed all terms that already had a zero exponent of L will be discarded

and other terms will be retained. The exponents of L in all the remaining terms will then be reduced by the lowest exponent of L. Each kernel expression contains all possible terms with zero exponent of L at that step, as well as all terms with higher exponents of L. Therefore if an instance of the common subexpression belongs to a polynomial, it will be a part of one of the kernel expressions of the polynomial. Both instances of the common subexpression will be a part of some kernel expressions and an intersection in the set of kernels will detect the common subexpression.

If the common subexpression does not satisfy the definition of a divisor, which means it does not have any term having a zero exponent of L, the subexpression obtained through division by the smallest exponent of L will also be a common subexpression, and will satisfy the definition of a divisor. Reasoning as above, this common subexpression will be detected by an intersection among the set of kernels.

3.c Matrix transformation to find kernel intersections

The set of kernels generated is transformed into a matrix form called the Kernel Intersection Matrix (KIM), to find kernel intersections. There is one row for each kernel generated and one column for each distinct term in the set of kernel expressions. Each term is distinguished by its sign (+/-), variable and the exponent of L.

		1	2	3	4	5	6
		$+X_1$	$+X_1L^2$	$+X_2$	$+X_2L$	$+X_2L^2$	$+X_1L$
1	1	$1_{(1)}$	$1_{(2)}$	$1_{(3)}$	$1_{(4)}$	$1_{(5)}$	
2	L			$1_{(4)}$	$1_{(5)}$		$1_{(2)}$
3	L^2	$1_{(2)}$		$1_{(5)}$			
4	L^2	$1_{(6)}$		$1_{(7)}$	$1_{(8)}$		

Figure 6. KIM for the kernels in Figure 4

The above figure shows the KIM for our example linear system from its set of kernels and co-kernels shown in Figure 4. The rows are marked with the co-kernels of the kernels which they represent. Each '1' element (i,j) in the matrix represents a term in the original set of expressions which can be obtained by multiplying the co-kernel in row i with the kernel term in column j. The number in parenthesis represents the term number that the element represents.

Each kernel intersection appears in the matrix as a rectangle. A **rectangle** is defined as a set of rows and columns such that all the elements are '1'. For example in the matrix in Figure 6, the row set {1,4} and the column set {1,3,4} together make a rectangle. A **prime rectangle** is defined as a rectangle that is not contained in any other rectangle.

The **value** of a rectangle is defined as the number of additions saved by selecting that rectangle (kernel intersection) as a common subexpression and is given by

$$Value(R,C) = (R-1)*(C-1) \qquad (II)$$

where R is the number of rows and C is the number of columns of the rectangle. The value of the rectangle is calculated after removing the appropriate rows and columns to remove overlapping terms. The algorithm for finding a Maximal Irredundant Rectangle (MIR) is explained in the next section.

We need to find the best set of common subexpressions (rectangles) such that the number of additions/subtractions is a minimum. A similar problem is encountered in multi-level logic synthesis [9, 10], where the least number of literals in a set of Boolean expressions is obtained by a minimum weighted rectangular covering of the matrix, which is NP hard [11]. Our problem is different since we have to deal with overlapping rectangles in our matrix. We use a greedy algorithm, where in each iteration, we pick the best non-overlapping prime rectangle.

Picking only non-overlapping prime rectangles need not always give the best results, since sometimes it may be beneficial to select a rectangle with overlapping terms and then subtract the terms that are covered more than once. We would like to consider the effect of selecting overlapping rectangles and observe its effect on the quality of the result in the future.

Worst case analysis of KIM: In the worst case, we have every possible term forming the columns in the KIM, and every possible exponent of L for each expression forming the rows. Consider an m x m constant matrix with N bits of precision. Since there are m variables and N possible exponents of L, and 2 possible signs (+/-), the worst case number of columns is 2*m*N. For the number of rows, consider the co-kernels generated for each expression. The exponents of L can range from 0 to N-1 and hence there are N possible kernels for each expression. Since there are m expressions, the worst case number of rows is m*N. The worst case number of prime rectangles happens in a square matrix where all the elements in the matrix are 1 except the ones on one diagonal [11], and the number of prime rectangles is exponential in the number of rows/columns of the square matrix. Therefore in our KIM transformation, the worst case number of prime rectangle and the subsequent time complexity to find the best prime rectangle is of $O(2^{mN})$

4. ELIMINATING COMMON SUBEXPRESSIONS

Since an exhaustive enumeration of all kernel intersections is not practical for real life examples, we consider heuristic algorithms to find the best set of kernel intersections. In this section we explain the heuristics and also an algorithm to handle the overlapping terms in a kernel intersection.

4.a Iterative algorithm to find kernel intersections

The algorithm for extracting kernel intersections is shown in Figure 7. It is a greedy iterative algorithm where the best prime rectangle is extracted in each iteration using a ping pong algorithm. In the outer loop, kernels and co-kernels are extracted from the set of expressions $\{P_i\}$ and the KIM is formed from that. The outer loop exits if there is no favorable kernel intersection in the KIM. Each iteration in the inner loop selects the most valuable rectangle if present based on the Value function in Equation II. The KIM is then updated by removing those 1's in the matrix that correspond to the terms covered by the selected rectangle.

Since the extracted prime rectangle (kernel intersection) may contain overlapping terms, we need to remove rows and/or columns from this rectangle to remove the overlapping terms. The algorithm for the extraction of MIR (ExtractMIR) is shown in Figure 7. The algorithm iteratively removes a row or a column from the rectangle that produces the maximum reduction in the number of overlapping terms. As an illustration consider the rectangle in the KIM in Figure 6 comprising of the rows {1,2,4} and the columns {3,4}. This rectangle has overlapping terms since the term 4 is repeated in the rectangle. It can be observed that removal of either row 1 or row 2 will remove the overlap in the rectangle.

Consider the KIM for our example linear system shown in Figure 6. The procedure for ping_pong_row returns the prime rectangle R_1 consisting of the rows {1,4} and the columns {1,3,4}, which has a value 2 (Equation II). The procedure for ping_pong_col produces the rectangle comprising the rows {1,3,4} and the columns {1,3}, which also has a value 2. Rectangle R_1 which corresponds to the subexpression $D_1 = X_1 + X_2 + X_2L$ is chosen arbitrarily. This rectangle covers the terms {1,3,4,6,7,8}. The KIM is updated by removing these terms

covered by the selected rectangle. No more kernel intersections can be found in the KIM and the inner loop exits.

```
Find_Kernel_Intersections( {Pi}, {Xi})
{
  {Pi} = Set of polynomial expressions
  {Xi} = Original Variables

  while(1)
  {
     For (all expressions in {Pi} )
        {D} = {D} ∪ Kernels(Pi);
     KIM = Form Kernel Intersection Matrix({D});
     if( no favorable rectangle exists) return;

     {R} = Set of new kernel intersections = φ;
     {V} = Set of new variables = φ;
     while(favorable rectangle exists)
     {
       R1 = Best_Rectangle_Ping_Pong(KIM);
       MIR_R1 = ExtractMIR(R1);
       {R} = {R} ∪ MIR_R1;
       {V} = {V} ∪ New Variable;
       Update KIM;
     }

     Rewrite {Pi} using {R};
     {Pi} = {Pi} ∪ {R};
     {Xi} = {Xi} ∪ {V};
  }
}
```

```
ExtractMIR(R)
{
  R = Rectangle;
  while(R has at least 2 rows and 2 columns)
  {
    if(R does not have overlapping terms)
    return R;

   Remove row or column that reduces the most
   number of overlapping terms from R;
  }

return φ; // could not find a useful rectangle
}


Best_Rectangle_Ping_Pong(KIM)
{
  seed_row = seed row in KIM;
  Rectangle R1 = ping_pong_row(seed_row);

  seed_col = seed column in KIM;
  Rectangle R2 = ping_pong_col(seed_col);

  if( value(R2) > value(R1) ) return R2
  else return R1
}
```

Figure 7. Algorithm to find kernel intersections

The expression for D_1 is added to the set of expressions and a new variable D_1 is added to the set of variables. The expressions are rewritten as shown in Figure 8. The KIM is constructed for these expressions, and is shown in Figure 9.

$$Y_1 = {}_{(1)}D_1 + {}_{(2)}X_1L^2 + {}_{(3)}X_2L^2$$
$$Y_2 = {}_{(4)}D_1L^2$$
$$D_1 = {}_{(5)}X_1 + {}_{(6)}X_2 + {}_{(7)}X_2L$$

Figure 8. Expressions after 1st Iteration

There is only one valuable rectangle in this matrix that corresponds to the rows {2,3} and the columns {4,5}, and has a value 1. This rectangle is selected. No more rectangles are selected in the further iterations and the algorithm terminates. The final set of expressions is shown in Figure 1d.

		1	2	3	4	5	6
		$+D_1$	$+X_1L^2$	$+X_2L^2$	$+X_1$	$+X_2$	$+X_2L$
1	1	$1_{(1)}$	$1_{(2)}$	$1_{(3)}$			
2	L^2				$1_{(2)}$	$1_{(3)}$	
3	1				$1_{(5)}$	$1_{(6)}$	$1_{(7)}$

Figure 9. Extracting Kernel Intersections (2nd Iteration)

4.b Finding the best prime rectangle by using ping pong algorithm

The procedure for finding the best prime rectangle is similar to the ping pong algorithm used in multi-level logic synthesis[9, 12]. We do not give a detailed description of the algorithm here due to space limitations. The algorithm extracts two rectangles from the procedures ping_pong_row and ping_pong_col and selects the best one between them. The procedure ping_pong_row tries to build a rectangle, starting from a seed row, each time adding a row by intersecting the column set, till the number of columns becomes less than 2. The criterion for selecting the best row in each step is based on the value of the maximum irredundant rectangle (MIR) that can be extracted from the rectangle created by intersection with that row. The rectangle with the best value (the one that saves the most number of additions/subtractions) is recorded during this construction and is returned at the end of the procedure. This algorithm is quadratic in the number of rows in the matrix. The algorithm for ping_pong_col follows the same steps as ping_pong_row, except that the rectangle is built by intersecting with a column in each step, and the algorithm is quadratic in the number of columns in the KIM.

5. EXPERIMENTAL RESULTS

The goal of our experiments was to compare the reduction of additions/subtractions achieved by our technique with conventional techniques for common subexpression elimination and observe its effect on the area and latency of the synthesized hardware, for the matrix form of linear systems. For our experiments we considered Discrete Cosine Transform (DCT), Inverse Discrete Cosine Transform (IDCT), Discrete Fourier Transform (DFT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT). For DFT, the matrix was split into a real part (RealDFT) and an imaginary part (ImagDFT) and the optimizations were carried out separately for the two matrices. The experimental results were done for 8x8 constant matrices (8 point). The constants in all these examples were represented using 16 digits of precision.

There are a number of different techniques for finding common subexpressions in multiple constant multiplications [1, 2, 4, 6-8], and it is not possible to compare our results with all of them. We compared our algorithm with the Recursive Shift-and-Add Decomposition for Sharing (**RESANDS**) algorithm [4], which finds common subexpressions among all shifted forms of constants multiplied by a single variable. We do not consider the matrix splitting suggested in [4], and compare only with the RESANDS algorithm since the objective is to compare different techniques for common subexpression elimination. We chose this work since it also targets matrix form of linear systems. We also compared our results with the results of Potkonjak et al. [1] since it is the most widely referenced work on the multiple constant multiplication problem. In [1, 5] a post processing step is suggested for matrix form of linear systems where each expression after the first stage of optimization is viewed as a bit pattern and the number of additions are reduced by finding common bit patterns. We also implemented this step to make a fair comparison with our work. We compared the number of additions/subtractions for the Canonical Signed Digit (CSD) representation of the constants since this representation gives an implementation with the least number of additions compared to other representations [13]. From Table 1, it is clear that our technique always gives an implementation with the minimum number of additions/subtractions compared to the other techniques. The results of RESANDS is better than that of Potkonjak et al, since RESANDS is able to find common subexpressions even among shifted forms of the constants. Our algorithm can do these optimizations in addition to finding common subexpressions involving multiple variables.

Table 1. Comparing number of additions/subtractions (CSD representation of constants)

Example	Number of additions/subtractions				% Improvement over		
	Original (I)	RESANDS (II)	Potkonjak (III)	Our Technique (IV)	(I)	(II)	(III)
DCT	274	202	227	174	36.5	13.1	23.3
IDCT	242	183	222	162	33.0	11.5	27.0
RealDFT	253	193	208	165	34.8	14.5	20.7
ImagDFT	207	178	198	134	35.3	24.7	32.3
DST	320	238	252	200	37.5	16.0	20.6
DHT	284	209	211	175	38.4	16.3	17.0
Average	**699.2**	**493.2**	**569.2**	**413.5**	**41.7**	**16.2**	**28.3**

We synthesized the designs using Synopsys Design Compiler™ and Synopsys Behavioral Compiler™ using the 1.0 μm CMOS standard.db technology library using a clock period of 50 ns. We used the Synopsys DesignWare™ library for our functional units. The designs were scheduled with minimum latency constraints. With this constraint, the tool tries to minimize the latency as much as possible assuming unlimited resources, and then tries to reduce the hardware for the achieved minimum latency. The area and latency of the synthesized examples are shown in Table 2 and Table 3, respectively.

Table 2. Area of synthesized examples

Example	Area (Library Units)				% Improvement over		
	Original (I)	RESANDS (II)	Potkonjak (III)	Our Technique (IV)	(I)	(II)	(III)
DCT	102366	90667	96375	73311	28.4	19.1	23.9
IDCT	105345	81868	99771	66864	36.5	18.3	33.0
RealDFT	102615	90496	84770	69827	31.9	23.2	17.6
ImagDFT	93839	75140	84864	55940	40.4	25.5	34.1
DST	120090	108101	106498	84715	29.5	21.6	20.4
DHT	111118	93939	79409	71272	35.9	24.1	10.2
Average	**105896**	**90110**	**91948**	**70322**	**33.8**	**22.0**	**23.2**

Table 3. Latency of synthesized examples

Example	Latency (clock cycles)				% Improvement over		
	Original (I)	RESANDS (II)	Potkonjak (III)	Our Technique (IV)	(I)	(II)	(III)
DCT	10	10	11	11	-10.0	-10.0	0.0
IDCT	9	10	11	11	-22.2	-10.0	0.0
RealDFT	9	10	12	11	-22.2	-10.0	8.3
ImagDFT	9	10	11	10	-11.1	0.0	9.1
DST	10	11	12	11	-10.0	0.0	8.3
DHT	9	11	11	11	-22.2	0.0	0.0
Average	**9.3**	**10.3**	**11.3**	**10.8**	**-16.2**	**-5.0**	**4.3**

From the results it is clear that our technique gives the least area compared to the other methods, and produces an average of 22% lower area compared to the next best technique (RESANDS). The latency of our optimized implementations has increased by about 16.2% compared to the unoptimized case. This can be explained by the fact that these designs were scheduled with minimum latency constraints and unlimited resources. While the original expressions do not have to wait for the results of any common subexpressions, this is not the case for the optimized designs. Hence the optimized designs can have a longer critical path. It

should be noted that this reduced latency for the unoptimized case was achieved at the expense of 33.8% more area than our optimized implementation. In future we would like to modify our algorithm such that, the selection of common subexpressions can be controlled for different objective functions and constraints.

6. CONCLUSIONS

In this paper we presented a new transformation of constant multiplications and algorithms to find common subexpressions involving multiple variables, for the matrix form of linear systems. Our technique can be used to find common subexpressions in any kind of linear computations where there are a number of multiplications with constants involving any number of variables. Experimental results show that our technique gave an implementation with the minimum number of additions/subtractions over known techniques which primarily optimize multiplications with only a single variable at a time. Synthesis results on a subset of these examples showed an implementation with lesser area than conventional techniques.

Right now our algorithm only optimizes the number of operations in a given transform. In future we would like to extend our algorithm to perform optimizations for different objective functions such as area and latency, with given constraints.

REFERENCES

[1] 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.

[2] R.Pasko, P.Schaumont, V.Derudder, V.Vernalde, and D.Durackova, "A new algorithm for elimination of common subexpressions," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1999.

[3] I.-C. Park and H.-J. Kang, "Digital filter synthesis based on an algorithm to generate all minimal signed digit representations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 1525-1529, 2002.

[4] H.T.Nguyen and A.Chattejee, "Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, pp. 419-424, 2000.

[5] K.K.Parhi, *VLSI Digital Signal Processing Systems*: John Wiley & Sons, Inc, 1999.

[6] R.I.Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 43, pp. 677-688, 1996.

[7] H.Safiri, M.Ahmadi, G.A.Jullien, and W.C.Miller, "A new algorithm for the elimination of common subexpressions in hardware implementation of digital filters by using genetic programming," *Proceedings of the Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, 2000.

[8] M.Mehendale, S.D.Sherlekar, and G.Venkatesh, "Synthesis of multiplier-less FIR filters with minimum number of additions," *Proceedings of the International Conference on Computer Aided Design*, 1995.

[9] 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.

[10] 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.

[11] R.Rudell, "Logic Synthesis for VLSI Design," PhD Thesis, University of California, Berkeley, 1989.

[12] R.K.Brayton, G.D.Hachtel, C.T.McMullen, and A.S.Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*: Kluwer Academic Publishers, 1984.

[13] K.Hwang, *Computer Arithmetic: Principle, Architecture and Design*: Wiley, 1979.