Simultaneous Optimization of Delay and Number of Operations in Multiplierless Implementation of Linear Systems

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

Many optimization techniques exist for the hardware implementation of multiplierless linear systems. These methods mainly reduce the number of additions and subtractions by eliminating common subexpressions. These optimizations can increase the critical path of the computations, which is unacceptable for high performance systems.

In this work, we present a technique that controls the delay during the optimization. To the best of our knowledge this is the only technique that achieves the best possible delay and at the same time finds all possible common subexpressions. Furthermore our technique can be applied to linear systems consisting of any number of variables. We achieve an average of 50% fewer operations for the same delay, over a method that extracts only non-recursive common subexpressions.

Index Terms – *delay optimization, common subexpression elimination, DSP transforms, multiplierless filters, logic synthesis*

1. Introduction

Linear systems such as Finite Impulse Response (FIR) filters and Discrete Signal Processing (DSP) transforms are present in a number of applications such as wireless communication, audio and video processing, and digital television. These computations mainly consist of multiplication of a vector of input samples with a set of constant coefficients. Multiplications are expensive in terms of area and power consumption, when implemented in hardware. The relative cost of an adder and a multiplier in hardware, depends on the adder and multiplier architectures. For example, a *k x k* array multiplier has k times the logic (and area) and twice the latency of the slowest ripple carry adder.

Since the values of the coefficients are known beforehand, the full flexibility of a multiplier is not necessary, and it can be more efficiently implemented by converting it into a sequence of additions/subtractions and shift operations. A popular and well researched technique for implementing the transposed form of FIR filters is the use of a multiplier block, instead of using multipliers for each constant as shown in Figure 1b. The multiplications with the set of constants $\{h_k\}$ are replaced by an optimized set of additions and shift operations, involving computation sharing.

This idea of a multiplier block was first proposed in [3], and since then there have been a number of papers on optimizing the multiplier block using methods such as coefficient scaling and finding common computations [1, 4-10]. However, not much work has been done to control the delay of the optimized computations. In this work, we define delay as the number of additions in the critical path of the multiplierless implementation of the filters. Sharing of common computations can increase the critical path of the computations, slowing down the performance, which can be unacceptable for high speed applications. Therefore it is important to consider delay during optimizations.
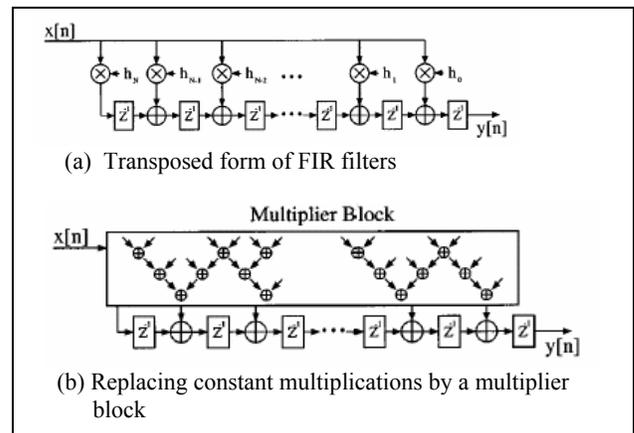

(a) Transposed form of FIR filters

(b) Replacing constant multiplications by a multiplier block

Figure 1. Multiplier block implementation of FIR filters

Figure 2 shows an example where sharing of computations increases the critical path. Figure 2a shows the computation of the expression F = a + b + c + d + (a + b + c)<<2. This expression has six additions and by using the fastest tree structure for this expression, we get a critical path of three additions. From the expression, we can see that there is a common computation (a + b + c). By extracting this common subexpression, we can reduce the number of additions by two, but the critical path is

increased to four adder steps as shown in Figure 2b. Figure 2c, shows the result of a more careful selection of common subexpressions. Here (a + b) is selected as a common subexpression, and though the savings in the number of additions is only one, the critical path does not increase. We use this example to show the working of our algorithm.

Thus it can be seen that delay can increase if common subexpressions are not extracted carefully. Most of the earlier works have concentrated on reducing the number of operations. In this paper we have modified common subexpression elimination algorithms to perform delay aware common subexpression elimination. Delay can be easily controlled by using simple optimization algorithms such as the Non-Recursive Common Subexpression Elimination (NRCSE) algorithm [1], but the reduction in the number of operations is significantly reduced, as we show in our experimental results section.
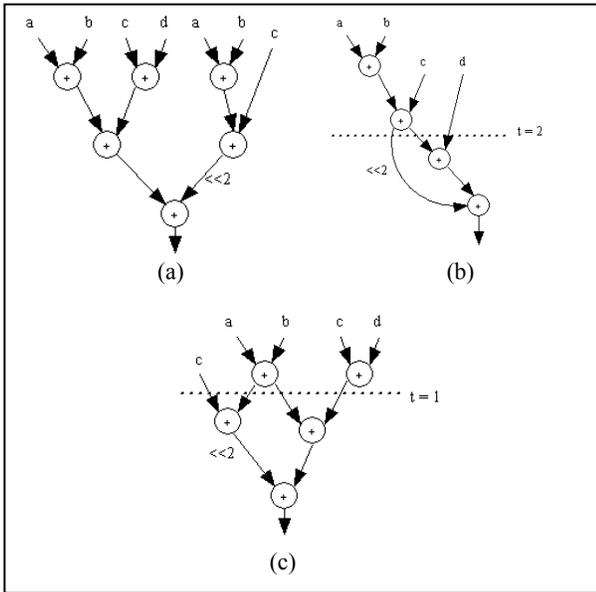
Figure 2. Optimization example a) Original expression tree b) Eliminating common subexpressions c) Delay aware common subexpression elimination

The techniques for common subexpression elimination that we show in this paper are based on the Fast Extract (FX) algorithm [11, 12]used for Boolean expression optimization. We have adapted these techniques to optimize arithmetic expressions consisting of addition, subtraction and shift operators. The rest of the paper is organized as follows. Section 2 introduces the terminologies used in this paper and formulates the problem that is solved. Section 3 presents some related work on multiplierless implementation of linear systems, and compares their contributions with ours. Section 4 presents some background work on the algebraic methods used for eliminating common subexpressions. We present

the delay aware optimization technique in Section 5. In Section 6, we present experimental results and finally conclude the paper in Section 7.

2. Problem formulation

In this section we introduce the terminology used in the paper and formulate the problem to be solved.

CSD representation: CSD (Canonical Signed Digit) representation of a constant is a signed digit representation, using the digit set {1,0,-1}. This representation gives the minimum number of non-zero digits [13] because of which it is widely used in multiplierless implementations. For example, consider the implementation of 7*X. Using two's complement representation for the constant, this can be implemented as X + X<<1 + X<<2. Using CSD representation, this can be represented using one fewer addition as X<<3 – X.

Minimum latency for a linear system: Let us assume that we are only interested in the fastest tree implementation of the set of the additions and subtractions of a linear system. We assume that we have enough adders to achieve the fastest tree structure and achieve the minimum possible latency. The minimum latency of the system is then determined by the expression having the maximum number of terms. Let N_{max} be the number of terms in the longest expression. The latency of the system is then given by

$$\text{Min-Latency} = \lceil \log_2 \mathbf{N}_{max} \rceil \quad (I)$$

For the expression in Figure 2, which has seven terms, the minimum latency is $\lceil \log_2 \mathbf{7} \rceil$ = 3. This is the minimum possible latency, for the linear system, and is obtained without computation sharing. Sharing of computations will decrease the number of additions/subtractions, but it cannot decrease the length of the critical path. In fact sharing can increase the length of the critical path, as was illustrated in Figure 2.

For the case of the multiplier block implementation of FIR filters (as shown in Figure 1), the minimum latency is governed by the constant coefficient having the most number of non-zero digits K_{max} in its CSD representation. The minimum latency is then given by

$$\text{Min-Latency} = \lceil \log_2 (\mathbf{K}_{max} - \mathbf{1}) \rceil \quad (II)$$

Recursive and Non-Recursive Common Subexpression Elimination: A recursive common subexpression is a subexpression that contains at least one other common subexpression extracted before. For example, consider the constant multiplication (1010-101010-1)*X as shown in Figure 3a. The common subexpression $d_1 = X + X<<2$ (Figure 3b) is non-recursive, and it reduces the number of

additions by one. Now, the common subexpression $d_2 = (d_1 <<2 - X)$ is recursive since it contains the variable d_1 which corresponds to a previously extracted common subexpression. Extracting this leads to the elimination of one more addition.

Problem statement: The problem that we are targeting in this paper can be stated thus. Given a multiplierless realization of a linear system, minimize the number of additions/subtractions as much as possible such that the latency does not exceed the minimum specified latency. In this work we constrain this latency to the minimum possible latency, as described by equations I and II. As per the problem statement, we try to eliminate as many additions as possible by exploring even recursive common subexpression elimination.

F = (1010-101010-1)*X
  = X<<10 + X<<8 - X<<6 + X<<4 + X<<2 – X
                (a) Original expression

d_1 = X + X<<2
F = d_1<<8 + d_1<<2 – X<<6 – X
   (b) Non-recursive common subexpression elimination

d_1 = X + X<<2
d_2 = d_1<<2 – X
F = d_2<<6 + d_2
     (c) Recursive common subexpression elimination

Figure 3. Recursive and Non-Recursive common subexpression elimination

3. Related Work

There have been a number of works on the multiplierless realization of linear systems particularly for the optimization of the multiplier block of the FIR digital filters (shown in Figure 1) [1, 3-10, 14]. These works are based on finding common digit patterns in the coefficients, since they are all used to multiply only a single variable at a time. These techniques range from graph based coefficient synthesis techniques [3-6], to exhaustive enumeration of all possible digit patterns [8, 14]. There has also been some work on matrix form of linear systems, which involve multiple variables [2, 15-17]. The optimizations in these works are performed by matrix splitting to modify the constants [15, 16] and performing common subexpression elimination. Despite the large number of techniques for redundancy elimination, unfortunately, there are not many methods developed for controlling the critical path, which is essential in the design of high speed systems.
We found one technique that promised the realization of filters with the given delay constraints, described in [7, 18]. The authors make use of the methods BHM (Modified Bull Horrocks method [4]) and the RAGn (n-dimensional Reduced Adder Graph Method [3] to obtain reduction in

the number of operations. They then use a set of transformations in an iterative loop to reduce the delay. If these transformations are not able to decrease the delay, then the algorithm reverts back to the original CSD representation. These methods use graph synthesis methods for finding common computations, by using partial sums common in coefficients to implement other coefficients. These methods are not suited for delay optimization because they create a long chain of dependencies. The delay improvement proposed in [7], uses an iterative improvement where a number of optimizations are reversed. In contrast to this backtracking algorithm, our method controls delay during the processing of selecting common subexpressions. Furthermore, our algorithm can handle linear systems involving multiple variables, and can thus be used for a wider range of linear systems.
In the work done by [1], the authors use a two-term non-recursive common subexpression elimination algorithm for reducing the number of operations. They restrict the type of subexpressions to the type shown in Figure 3b, with each subexpression restricted to only two terms. It is obvious that such a policy does not increase the critical path of the computations, but this produces limited reduction in the number of operations, since the savings produced by recursive common subexpressions is not explored. For example, the optimizations shown in Figure 3b and Figure 3c both have a critical path of three adder steps, but the use of recursive common subexpressions leads to a saving of one more addition.

4. Common Subexpression Elimination

In this section, we briefly describe the common subexpression elimination algorithm, on which the proposed work is based on. We use a polynomial transformation of linear systems, which allows us to optimize multiple variable systems. Subexpression elimination is achieved using a greedy iterative algorithm, where the best two-term common subexpression is extracted in each iteration. A detailed description of this extraction algorithm can be found in [2].

4.1 Polynomial transformation of linear systems
Using a given representation of the constant C, the multiplication with the variable X (assuming only fixed point representation) can be represented as

$$C*X = \sum_i XL^i \quad \text{(III)}$$

The exponent of L represents the magnitude of the left shift and the i's represent the digit positions of the non-zero digits of the constants. For example the multiplication $7*X = (100-1)_{CSD}*X = X<<3 – X = XL^3 – X$, using the polynomial transformation. Consider the example expression used in Figure 2, which is reproduced in Figure

5a. Using the polynomial transformation, the expression is rewritten as shown in Figure 5b.

4.2 Concept of two-term divisors

Common subexpressions are found by intersections among the set of two-term divisors (called only divisors from now on). Divisors are obtained from every pair of terms in an expression by dividing by the minimum exponent of L. For example consider the expression $P = a + bL^2 + cL^3$. Consider the two terms $\{+bL^2, +cL^3\}$. The minimum exponent of L in these two terms is L^2. Dividing by L^2 gives the divisor $(b + cL)$. The number of divisors in an expression with N terms is $\binom{N}{2}$.

Finding intersections among these set of divisors can extract the two-term common subexpressions. Two divisors are said to **intersect** if they are derived from disjoint terms and they are the same, with or without reversing their signs. Therefore by this definition, the divisors $X_1 - X_2$ and $X_2 - X_1$ are said to intersect, if they are derived from disjoint terms. Two divisors are said to **overlap** if at least one of the terms from which they are obtained is common.

F = a + b + c + d + a<<2 + b<<2 + c<<2
    (a) Expression used in Figure 2.

$F = a + b + c + d + aL^2 + bL^2 + cL^2$
    (b) Using polynomial transformation

Figure 5. Polynomial transformation for example expression

These divisors are quite significant since all common subexpressions can be found by an intersection among the set of divisors, as is illustrated by the following theorem.

Theorem: There exists a multiple term common subexpression in a set of expressions *if and only if* there exists a non-overlapping intersection among the set of divisors of the expressions.

A detailed proof of this theorem can be found in [2].

5. Delay aware common subexpression elimination

In this section, we present the algorithm for delay aware common subexpression elimination. This algorithm is based on the algebraic method described in Section 4, which is detailed in [2]. The algorithm takes into account the effect of delay on selecting a particular divisor as a common subexpression. Only those instances of a divisor that do not increase the delay of the expression beyond the maximum specified delay limit are considered. We first describe how the delay of an expression on selection of divisor instances can be calculated. We then explain the main algorithm.

5.1 Calculating delay

Each divisor is associated with a **level** which represents the delay in terms of the number of adder steps involved in computing the divisor. Each divisor is also associated with the number of original terms that are covered by it. This is illustrated in Figure 6.

The procedure for calculating the delay of an expression, after selecting a divisor D, is outlined in Figure 7. The terms $\{T_E\}$ of the expression are partitioned into the terms $\{T_1\}$ covered by the divisor and the remaining terms $\{T_2\}$.

F = a + b + c + d

d_1 = (a + b) Level(d_1) = 1
Original terms covered (d_1) = 2

$d_2 = d_1 + c$ Level(d_2) = 2
Original terms covered(d_2) = 3

Figure 6. Divisor information

p = # of instances of Divisor D in expression
t = Delay(adder-steps) in computing divisor D

$\{T_1\}$ = current terms covered by 'p' instances of D
$\{T_E\}$ = current terms in the expression
$\{T_2\} = \{T_E\} - \{T_1\}$ = Remaining terms

K = # of Values in $\{T_2\}$ still available for computation after time t

Total values available = p + K
Delay of expression = $(t + \lceil \log_2(p + K) \rceil)$

Figure 7. Procedure for calculating delay

The delay is calculated from the number of values that are available for computation after the time (t) taken to compute the divisor under investigation. Among $\{T_1\}$ terms, there will be 'p' values available corresponding to the 'p' instances of the divisor. We need to find the number of values from $\{T_2\}$ that are available after time t. In general, we need to schedule the terms in $\{T_2\}$ to get this information. But scheduling for every candidate divisor using a simple algorithm like As Soon As Possible (ASAP), which is quadratic in the number of terms is expensive. For a lot of cases, we can estimate this number using a simple formula.

Let T_{2o} be the number of original terms corresponding to the terms in $\{T_2\}$. If none of the terms in $\{T_2\}$ have been covered by any divisor, or they are covered by divisors

covering power of two original terms implemented in the fastest tree structure (covering 2^j original terms with delay j), then K can be quickly calculated using the formula

$$K = \left\lceil \frac{T_{2o}}{2^t} \right\rceil \quad \textbf{(IV)}$$

The cases in which we can speedup the algorithm are:

1. The divisor covers power of 2 original terms with the fastest possible tree structure (2^j original terms with delay of j). In this case, we do not even need to estimate the delay, and all non-overlapping instances can be extracted without increasing the delay.
2. The remaining terms (terms not covered by the divisor) have not been covered by any divisor.
3. Of the remaining terms (terms not covered by the divisor), some or all of the terms may be covered by divisors. If these divisors cover power of 2 original terms with the fastest possible tree structure, then the formula can be used. Using these pruning conditions helps to significantly speed up the algorithm, as we report in the experimental results section. If the terms in $\{T_2\}$ do not satisfy this criterion, then K has to be calculated using ASAP (As Soon As Possible) Scheduling [19].

$F = a + b + c + d + aL^2 + bL^2 + cL^2$

$d_1 = (a + b) : delay = t = 1$
$p = 2$ instances of d_1 in F
$\{T_1\} = \{a,b, aL^2, bL^2\}$
$\{T_2\} = \{c,d,cL^2\}$
$K = 2$
$Delay = 1 + \lceil \log_2(2 + 2) \rceil = 3$

(a) Selecting $d_1 = (a + b)$

$F = d_1 + d_1L^2 + c + d + cL^2$
$d_2 = (d_1 + c) : delay(d_2) = t = 2$
$p = 2$
$\{T_1\} = \{d_1, c, d_1L^2, cL^2\}$
$\{T_2\} = \{d\}$
$K = 1$
$Delay = 2 + \lceil \log_2(2 + 1) \rceil = 4$

(b) Selecting $d_2 = (d_1 + c)$

Figure 8. Delay calculation for example expression

The delay calculation for our example expression is illustrated in Figure 8. In 8a, the delay calculation for divisor $d_1 = (a + b)$ is illustrated. The delay of this divisor is one adder step. The expression tree corresponding to the selection of this divisor is shown in Figure 2c. From this figure, we can see that four values are available for computation after one adder step. Two of them (p) correspond to the two uses of the divisor d_1 and K = 2 of them are from $\{T_2\}$ (the terms other than those covered by

d_1). K can also estimated by the formula in Equation IV. The delay is calculated to be 3.
Figure 8b shows the delay calculation when $d_2 = (d_1 + c)$ is selected. The delay of the divisor d_2 is two adder steps. The expression tree showing the selection of d_2 can be seen in Figure 2b. From the figure, the number of values available for computation after t = 2 adder steps is three. Two of them (p) correspond to the two uses of divisor d_2 and the other one (K) corresponds to the value d. K can also be calculated using equation IV. The delay is calculated to be 4, and the expression tree is as shown in Figure 2b.

5.2 Optimization algorithm

The main algorithm is shown in Figure 9. The algorithm consists of two steps. In the first step, frequency statistics of all the distinct divisors are computed and stored. This is done by generating divisors $\{D_{new}\}$ for each expression and looking for intersections in the existing set $\{D\}$ of generated divisors. For every intersection, the frequency statistic of the matching divisor d_1 in $\{D\}$ is updated and the matching divisor d_2 in $\{D_{new}\}$ is added to the list of intersecting **instances** of d_1. The unmatched divisors in $\{D_{new}\}$ are then added to $\{D\}$ as distinct divisors.

Optimize ({P_i})
{
    {P_i} = Set of expressions in polynomial form;
    {D} = Set of divisors = φ ;
    // Step 1. Creating divisors and their frequency statistics
    for each expression P_i in {P_i}
    {
        *{D_new} = **Divisors(P_i)**;*
        Update frequency statistics of divisors in {D};
        {D} = {D} ∪ { D_new};
    }

    //Step 2. Iterative selection and elimination of best divisor
    MaxDelay = Maximum specified delay (adder steps) of
                    expressions

    while (useful divisor available)
    {
        Find d = Divisor in {D} having the most number of
            non-overlapping instances not increasing
            the critical path;

        Rewrite all expressions using d;
        Update divisors in {D};
    }
}

Figure 9. Algorithm for delay aware common subexpression elimination

In the second step of the algorithm, the best divisor is selected and eliminated in each iteration. We define the "best divisor" to be the divisor that has the most number of

non-overlapping instances that do not increase the delay of the expressions beyond the maximum specified value. This value known as the *true value* is calculated for each distinct candidate divisor. The procedure for calculating this *true value* is given in Figure 10. The set $\{d_{instances}\}$ contains the instances of the divisor d in the set of expressions. For each expression P_i that contains some of these instances, the set of valid instances $\{Valid\_d\}_i$ which do not increase the delay of the expression beyond MaxDelay, is computed. This is repeated and accumulated over all such expressions P_i.

The set of allowed instances $\{Allowed\_instances\}$ now contains the set of all non-overlapping instances of divisor d that do not increase the delay. This number of elements in this set represents the true value of the divisor.

After extracting the best divisor, the expressions are rewritten using the divisor. Some divisors from $\{D\}$ will be eliminated and some new divisors will be added, due to the rewriting of the expressions. The frequency statistics of the divisors will also change. All this is done dynamically in our algorithm.

For our example expression F shown in Figure 5, assume that the maximum specified delay MaxDelay is 3 adder steps, which is equal to the critical path of the expression. The divisor $d_1 = (a + b)$ has two instances in F. The delay of the expression F is calculated to be 3 adder steps, after selecting d_1 as a common subexpression (delay calculation is illustrated in Figure 8a.

Find_true_value(d , MaxDelay)
{

   d = distinct divisor in set of divisors {D}
   {d_{instances}} = Set of instances of divisor d
   {P_i} = Set of expressions containing
          any of the instances in {d_{instances}}
   {Allowed_instances} = φ;

   for each expression P_i in {P_i}
   {
     {Valid_d}_i = Set of non-overlapping instances
             that can be extracted in P_i without
             making its delay > MaxDelay;
     {Allowed_instances}
         = {Allowed_instances} ∪ {Valid_d}_i ;
   }

   True_value = |{Allowed_instances}| ;
   return True_value;
}

Figure 10. Algorithm to find the true value of a divisor

This divisor is the best divisor and is selected. After rewriting F (as shown in Figure 8b), the divisor $d_2 = (d_1 + c)$ is examined. This divisor has 2 instances in F, but the delay of the expression is increased to 4 adder steps by

choosing this divisor. Since the delay increases, it is not chosen, and the final expression is as shown in Figure 2c.

6. Experimental Results

In this section we present the results for our algorithm on a set of FIR filters and DSP transforms. We compared the delay produced by the common subexpression elimination algorithm without considering delay [2] with the delay aware common subexpression elimination algorithm presented in this paper. We also compared the number of additions and subtractions produced by the two algorithms and the Non-Recursive Common Subexpression Elimination (NRCSE) algorithm [1]. We first experimented with the following low-pass filters shown in Table 1. The filters are arranged according to the different filter specifications.

Table 1. Filter specifications

Filter #	Type	Order	f_p	f_s	R_p	R_s
1	BT	20	0.25	0.3	3	-50
2	EP	6	0.25	0.3	3	-50
3	LS	41	0.25	0.3	3	-50
4	PM	28	0.25	0.3	3	-50
5	BT	71	0.27	0.2875	2	-50
6	EP	8	0.27	0.2875	2	-50
7	LS	172	0.27	0.2875	2	-50
8	PM	119	0.27	0.2875	2	-50
9	EP	13	0.27	0.29	2	-100
10	LS	326	0.27	0.29	2	-100
11	PM	189	0.27	0.29	2	-100

There are 4 types of filters that we implemented: Butterworth (BT), Elliptical (EP), Least Square (LS), and Park Mc-Clennan (PM) filters. The orders represent the number of taps in the filters, which is the number of multiplications that need to be replaced by a multiplier block. f_p and f_s represent the passband and the stopband frequencies respectively. R_p and R_s represent the passband ripple and the stopband ripple (in dB) respectively. Table 2 gives the results for these filters in terms of the number of operations and critical path for the delay ignorant method [2], and the algorithm presented in this paper. All the coefficients in these filters are converted to integers and rounded, and are represented using 24 digits of precision.

Table 2 shows the number of additions (A) and critical path in terms of number of adder steps (CP) for the different optimization algorithms. From the above table, it can be seen that the algorithm that only considers common subexpression elimination exceeds the critical path in many cases. The delay aware optimization algorithm presented in this paper controls the delay in every step of the algorithm and therefore does not exceed the critical path. Due to the restrictive selection of common subexpressions this algorithm has **7.4%** more additions/subtractions over [2], but we still reduce the number of additions/subtractions by **70%** over the original representation. The table also shows the optimization done

by the non-recursive common subexpression elimination algorithm (NRCSE). Though the critical path is not exceeded in any case, it can be seen that the number of additions can be further reduced by **50.3%** on an average by exploring recursive common subexpressions, as is done by our method. The average run time of our delay aware optimization algorithm was only **0.1s**.

Table 2. Experiments on FIR filters

Filter #	Original		NRCSE [1]		[2]		This paper	
	A	CP	A	CP	A	CP	A	CP
1	106	4	59	4	24	4	24	4
2	158	4	102	4	46	5	46	4
3	238	4	146	4	67	6	74	4
4	26	3	16	3	10	3	10	3
5	234	4	129	4	60	5	60	4
6	656	4	378	4	168	5	186	4
7	868	4	527	4	217	6	248	4
8	44	4	27	4	14	4	14	4
9	988	4	652	4	281	7	334	4
10	1414	4	917	4	374	6	444	4
11	82	4	49	4	24	5	26	4
Average	437.6	3.9	273	3.9	116.8	5.1	133.3	3.9

We also experimented with a few common DSP transforms the Discrete Cosine Transform (DCT), Inverse Discrete Cosine Transform (IDCT), Discrete Fourier Transform (DFT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT) [20]. We considered the case where these transforms have 16x16 constant matrices (16-point transforms). Each coefficient has been quantized to 24 digits of precision. Table 3 shows the results for the multiple variable case.

Table 3. Experiments on DSP transforms (multi-variable)

Transform	Original		[2]		This paper	
	A	CP	A	CP	A	CP
DCT	2033	8	946	8	946	8
IDCT	1784	7	861	**8**	923	7
DFT	1761	7	853	**8**	925	7
DST	2066	8	949	8	949	8
DHT	2174	8	1013	8	1013	8
Average	1963.6	7.6	924.4	8	951.2	7.6

Here A represents the number of additions and CP represents the critical path in the number of adder steps. We did not compare the results produced by the NRCSE algorithm [1], since that algorithm only optimizes FIR filters, where only a single variable is involved. From the above table, it can be seen that the algorithm that only extracts common subexpressions produces expressions with increased critical path in some cases. The delay aware algorithm controls the critical path, with minimal difference in the number of operations compared to the

algorithm in [2]. The delay aware algorithm still manages to reduce the original operation count by **51.3%.** The average run time of our delay aware optimization algorithm was **298.6s** for these examples.

Speedup achieved by using formula in Equation IV

The formula in equation IV is used to reduce the number of calls to the ASAP scheduling algorithm, for the cases in which it can be applied. We also implemented the algorithm where we used ASAP scheduling for all cases, and compared the run times. The average run time in this case, for the examples in Table 3 is **1197s**. Therefore we achieve a speedup of **4 times** by using the formula.

7. Conclusions

In this paper, we present an algorithm that concurrently reduces the number of operations in multiplierless implementations of linear systems and at the same time controls the critical path. This is achieved by using a method that only considers two-term common subexpressions, which allows us to consider delay during the selection process. Our algorithm considers aims to minimize as many additions as possible by exploring even recursive common subexpressions. Another advantage of this algorithm is that it can be used to optimize linear systems consisting of any number of variables. Experimental results have shown that this algorithm can control the critical path of the computations with marginal increase in the number of additions/subtractions compared to the algorithm that only considers operation reduction.

Acknowledgements

References

[1]   M. Martinez-Peiro, E. I. Boemo, and L. Wanhammar, "Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 49, pp. 196-203, 2002.

[2]   A.Hosangadi, F.Fallah, and R.Kastner, "Reducing Hardware Compleity of Linear DSP Systems by Iteratively Eliminating Two Term Common Subexpressions," presented at Asia South Pacific Design Automation Conference, Shanghai, 2005.

[3]   D. R. Bull and D. H. Horrocks, "Primitive operator digital filters," *Circuits, Devices and*

Systems, IEE Proceedings G, vol. 138, pp. 401-412, 1991.

[4] A. G. Dempster and M. D. Macleod, "Use of minimum-adder multiplier blocks in FIR digital filters," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 42, pp. 569-577, 1995.

[5] H. Choo, K. Muhammad, and K. Roy, "Complexity reduction of digital filters using shift inclusive differential coefficients," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 52, pp. 1760-1772, 2004.

[6] K. Muhammad and K. Roy, "A graph theoretic approach for synthesizing very low-complexity high-speed digital filters," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 204-216, 2002.

[7] H.-J. Kang, H. Kim, and I.-C. Park, "FIR filter synthesis algorithms for minimizing the delay and the number of adders," presented at Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on, 2000.

[8] 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.

[9] A. P. Vinod and E. M.-K. Lai, "On the implementation of efficient channel filters for wideband receivers by optimizing common subexpression elimination methods," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, pp. 295-304, 2005.

[10] 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.

[11] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, pp. 778-793, 1992.

[12] J. Vasudevamurthy and J. Rajski, "A method for concurrent decomposition and factorization of Boolean expressions," presented at Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, 1990.

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

[14] 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.

[15] H.Nguyen and A.Chatterjee, "OPTIMUS: a new program for OPTIMizing linear circuits with number-splitting and shift-and-add decomposition," presented at Sixteenth International Conference on Advanced Research in VLSI, 1995.

[16] H.T.Nguyen and A.Chatterjee, "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.

[17] A.Hosangadi, F.Fallah, and R.Kastner, "Common Subexpression Involving Multiple Variables for Linear DSP Synthesis," presented at IEEE International conference on Application Specific Architectures and Processors (ASAP), Galveston, TX, 2004.

[18] H.-J. Kang and I.-C. Park, "FIR filter synthesis algorithms for minimizing the delay and the number of adders," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 48, pp. 770-777, 2001.

[19] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*: McGraw-Hill, Inc, 1994.

[20] S.K.Mitra, *Digital Signal Processing: A computer based approach*, second ed: McGraw-Hill, 2001.