Optimization Algorithms for the Multiplierless Realization of Linear Transforms

LEVENT AKSOY, INESC-ID
EDUARDO COSTA, Universidade Católica de Pelotas
PAULO FLORES and JOSE MONTEIRO, INESC-ID/IST TU Lisbon

This article addresses the problem of finding the fewest numbers of addition and subtraction operations in the multiplication of a constant matrix with an input vector—a fundamental operation in many linear digital signal processing transforms. We first introduce an exact common subexpression elimination (CSE) algorithm that formalizes the minimization of the number of operations as a 0-1 integer linear programming problem. Since there are still instances that the proposed exact algorithm cannot handle due to the NP-completeness of the problem, we also introduce a CSE heuristic algorithm that iteratively finds the most common 2-term subexpressions with the minimum conflicts among the expressions. Furthermore, since the main drawback of CSE algorithms is their dependency on a particular number representation, we propose a hybrid algorithm that initially finds promising realizations of linear transforms using a numerical difference method, and then applies the proposed CSE algorithm to utilize the common subexpressions iteratively. The experimental results on a comprehensive set of instances indicate that the proposed approximate algorithms find competitive results with those of the exact CSE algorithm and obtain better solutions than the prominent, previously proposed, heuristics. It is also observed that our solutions yield significant area reductions in the design of linear transforms after circuit synthesis, compared to direct realizations of linear transforms.

Categories and Subject Descriptors: B.2.0 [**Hardware**]: Arithmetic and Logic Structures—*General*; J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms: Algorithms, Design

Additional Key Words and Phrases: Constant matrix-vector multiplication, common subexpression elimination, 0-1 integer linear programming, numerical difference method

ACM Reference Format:
Aksoy, L., Costa, E., Flores, P., and Monteiro, J. 2012. Optimization algorithms for the multiplierless realization of linear transforms. ACM Trans. Des. Autom. Electron. Syst. 17, 1, Article 3 (January 2012), 27 pages.
DOI = 10.1145/2071356.2071359 http://doi.acm.org/10.1145/2071356.2071359

1. INTRODUCTION

The multiplication of constant(s) by a variable is a ubiquitous and crucial operation that has a significant impact on the design of Digital Signal Processing (DSP) systems, such as Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) filters, Fast Fourier Transforms (FFT), and Discrete Cosine Transforms (DCT). For example,

This work was supported in part by the Portuguese Foundation for Science and Technology (FCT) research project Multicon - Architectural Optimization of DSP Systems with Multiple Constants Multiplications PTDC/EIA-EIA/103532/2008 and by FCT through the PIDDAC Program funds.
Authors' addresses: L. Aksoy, P. Flores, and J. Monteiro, INESC-ID, Rua Alves Redol, 9, 1000-029, Lisboa, Portugal; email: levent@algos.inesc-id.pt; E. Costa, Universidade Católica de Pelotas, Rua Félix da Cunha, 412, Centro, Pelotas-RS, Brazil.

DOI 10.1145/2071356.2071359 http://doi.acm.org/10.1145/2071356.2071359

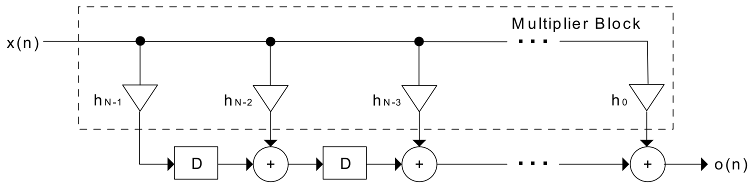


Fig. 1. Transposed form of a fully-parallel hardwired FIR filter implementation.

large numbers of single constant multiplications are required in the fast DCT method described in Hou [1987] and the implementation of a digital FIR filter in its transposed form needs the multiplication of filter coefficients by the filter input, as illustrated in Figure 1. In addition to its applications in DSP systems, the multiplication of constant(s) by a variable occurs frequently in cryptography applications, in the design of compilers, and in computer arithmetic.

A more general version of the multiplication of constant(s) by a variable is the multiplication of a constant matrix by an input vector. The Constant Matrix-Vector Multiplication (CMVM) operation is described as $Y_{k\times 1} = T_{k\times m} \cdot X_{m\times 1}$, where X is an input vector, Y is an output vector, and T is a matrix of constants specifying how the outputs are obtained from the linear transformation of the inputs. The CMVM operation is a central operation and performance bottleneck in FIR filter banks [Dempster and Murphy 2000], IIR filters, linear DSP transforms [Boullis and Tisserand 2005], and in the implementation of error correcting codes [Potkonjak et al. 1996].

In DSP systems, the constant multiplications are generally realized in a shift-adds architecture where each constant multiplication is realized using addition/subtraction and shifting operations [Nguyen and Chatterjee 2000]. This is simply because the constants to be multiplied by variable(s) are determined by the DSP algorithms beforehand, and thus the full-flexibility of a multiplier is not necessary. Note also that the multiplication operation is expensive in terms of area, delay, and power dissipation in hardware. Although the relative cost of an adder and a multiplier in hardware depends on the adder and multiplier architectures, an $n \times n$ array multiplier has approximately n times the area and twice the latency of the slowest ripple carry adder.

Furthermore, the design of constant multiplications in a shift-adds architecture enables the sharing of common partial products among the constant multiplications, yielding significant reductions in area and power dissipation of the design. Thus, the optimization problem is defined as finding the fewest numbers of addition/subtraction operations that generate the constant multiplications, since shifts can be implemented using only wires in hardware without representing any area cost. In this problem, it is also assumed that an adder and a subtracter have the same cost, although an addition and subtraction operation occupies different amounts of area in hardware, as shown in Aksoy et al. [2007].

As a simple example of multiple constant multiplications by a variable, consider $29x$ and $43x$, as illustrated in Figure 2(a). The shift-adds implementations of the constant multiplications are presented in Figures 2(b)-(c). Observe that while the multiplier-less implementation without partial product sharing requires 6 operations (Figure 2(b)), the sharing of partial products $3x$ and $5x$ in both multiplications reduces the number of operations to 4 (Figure 2(c)). As an example of more general constant multiplications, consider the linear transforms $y_1 = 3x_1 + 11x_2$ and $y_2 = 5x_1 + 13x_2$ obtained as a result of the multiplication of a constant matrix by the input vector in Figure 3(a). Observe from Figures 3(b)-(c) that while the shift-adds implementation of linear transforms without subexpression sharing requires 8 operations, the sharing of

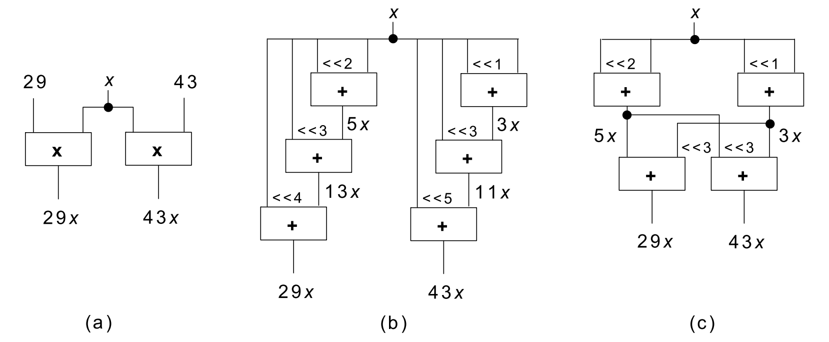


Fig. 2. (a) Constant multiplications of 29x and 43x; their shift-adds implementations: (b) without partial product sharing; (c) with partial product sharing.

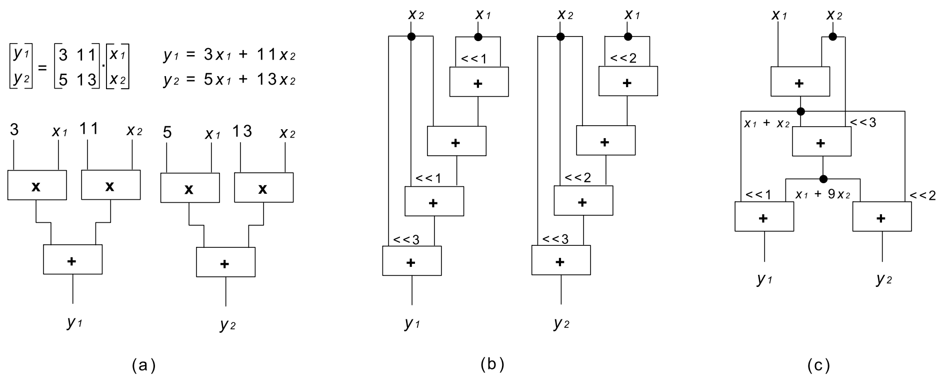

$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 3 & 11 \\ 5 & 13 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
$y_1 = 3x_1 + 11x_2$
$y_2 = 5x_1 + 13x_2$
$x_1 + x_2$
$x_1 + 9x_2$
<<1
<<2
<<3
y_1
y_2
(a)
(b)
(c)

Fig. 3. (a) Constant matrix-vector multiplication realizing $y_1 = 3x_1 + 11x_2$ and $y_2 = 5x_1 + 13x_2$; their shift-adds implementations: (b) without subexpression sharing; (c) with subexpression sharing.

subexpressions $x_1 + x_2$ and $x_1 + 9x_2$ in both transforms leads to an implementation with 4 operations.

The last two decades have seen a tremendous effort for the implementation of efficient algorithms designed for the multiplier-less realization of the constant multiplications by a variable [Aksoy et al. 2008, 2010; Dempster and Macleod 1995; Gustafsson et al. 2002; Hartley 1996; Lefevre 2001; Park and Kang 2001; Thong and Nicolici 2009; Voronenko and Püschel 2007]. The exact algorithms [Aksoy et al. 2008, 2010; Gustafsson et al. 2002] can be easily applied on real-size instances guaranteing the minimum number of operations solution; the heuristic algorithms [Dempster and Macleod 1995; Hartley 1996; Lefevre 2001; Park and Kang 2001; Thong and Nicolici 2009; Voronenko and Püschel 2007] can find solutions close to the minimum using little computational effort, and can deal with large-size instances that the exact algorithms cannot handle. However, for the multiplier-less realization of linear transforms using the fewest numbers of operations—to the best our knowledge—there is no exact algorithm that ensures the minimum solution. All existing algorithms are heuristics that cannot provide any indication on how far from the minimum their solutions are. Also, the proposed heuristics have generally been the modified versions of the algorithms designed for the multiplier-less realization of the constant multiplications by a variable.

In this article, we introduce an exact algorithm where all possible implementations of linear transforms are extracted when the constants are defined under a particular number representation. In the exact algorithm, the maximization of the subexpression sharing is realized by formulating the problem of finding the minimum number of subexpressions required to implement the linear transforms as a 0-1 Integer Linear Programming (ILP) problem and by finding the optimal solution using a generic 0-1 ILP solver. Due to the exponential growth in the size of the 0-1 ILP problem as the number of nonzero digits in the representation of a constant increases and the limitations of the 0-1 ILP solvers, there are still instances that the exact algorithm cannot handle. Hence, we also introduce a greedy heuristic algorithm that iteratively finds the most common 2-term subexpressions in the linear transforms, selects the one with the minimum conflicts, and implements it using a single operation. Furthermore, we propose a hybrid algorithm that iteratively finds alternative realizations of linear transforms using a numerical difference method [Muhammad and Roy 2002] and applies the heuristic algorithm to further reduce the complexity of design by sharing the common subexpressions. In this work, we also developed a tool that automatically generates the synthesizable VHDL code of direct implementation of linear transforms and their shift-adds implementations obtained by the high-level optimization algorithms for circuit synthesis.

The algorithms introduced in this article are compared with previously proposed algorithms on a comprehensive set of instances, including randomly generated constant matrices and linear DSP transforms. Experimental results show that the proposed approximate algorithms find solutions close to the minimum using little computational effort, compared to the solutions of the exact algorithm, and obtain significantly better solutions than those obtained by previously proposed algorithms. It is also observed that the design of linear transforms in a shift-adds architecture using the fewest numbers of operations leads to low-complexity and low-power circuits after its implementation at gate-level.

The rest of the article proceeds as follows. Section 2 gives the background concepts and Section 3 presents the problem definitions and an overview on previously proposed algorithms. Sections 4 and 5 introduce the exact and approximate algorithms, respectively. Experimental results are given in Section 6, and, finally, Section 7 concludes the article.

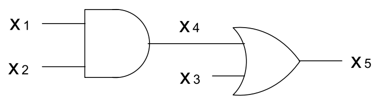
2. BACKGROUND

In this section, we provide basic notations on number representations, Boolean satisfiability, and the 0-1 ILP problem.

2.1 Number Representation

The *binary* representation decomposes a number in a set of additions of powers of two. The representation of numbers using a signed digit system makes use of positive and negative digits. Thus, an integer number represented in the *binary signed digit* system using n digits can be written as $\sum_{i=0}^{n-1} d_i 2^i$, where $d_i \in \{1, 0, -1\}$. Hereafter, we denote the digit -1 by $\overline{1}$. Observe that the binary signed digit system is a redundant number system, for example, both 0101 and $10\overline{1}\,\overline{1}$ correspond to the integer value 5.

The *Canonical Signed Digit* (CSD) representation [Avizienis 1961] is a signed digit system that has a unique representation for each number and verifies the following main properties: (i) two nonzero digits are not adjacent; (ii) the number of nonzero digits is minimal. Any n digit number in CSD has at most $\lceil (n+1)/2 \rceil$ nonzero digits and, on average, the number of nonzero digits is reduced by 33% when compared to the binary representation [Garner 1965]. This representation is widely used in multiplierless implementations of constant multiplications, because it reduces the hardware requirements due to the minimum number of nonzero digits. The *Minimal Signed Digit* (MSD) representation [Park and Kang 2001] is obtained by dropping the first property



$$\varphi = (x_1+\overline{x_4}).(x_2+\overline{x_4}).(\overline{x_1}+\overline{x_2}+x_4).$$
$$(\overline{x_3}+x_5).(\overline{x_4}+x_5).(x_3+x_4+\overline{x_5})$$

Fig. 4. A combinational circuit and its corresponding CNF formula.

of the CSD representation. Thus, a constant can have several representations under MSD, including the CSD representation of the constant, but all with a minimum number of nonzero digits.

As an example, suppose the constant 23 is defined in six bits. The representation of 23 in binary, 010111, includes 4 nonzero digits. The constant is represented as $10\overline{1}00\overline{1}$ in CSD, and both $10\overline{1}00\overline{1}$ and $01100\overline{1}$ denote 23 in MSD.

2.2 Boolean Satisfiability

A *propositional formula* denotes a Boolean function $\varphi : \{0, 1\}^n \rightarrow \{0, 1\}$. A *Conjunctive Normal Form* (CNF) is a representation of a propositional formula φ consisting of a conjunction of propositional clauses where each *clause* is a disjunction of literals, and a *literal* l_j is either a variable x_j or its complement $\overline{x_j}$. Note that if a literal of a clause assumes value 1, then the clause is satisfied. If all literals of a clause assume value 0, then the clause is unsatisfied. The *satisfiability* (SAT) *problem* is to find an assignment on n variables of the Boolean formula in CNF that evaluates the formula to 1 or to prove that the formula is equal to the constant 0.

A *combinational circuit* is a directed acyclic graph with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The *primary inputs* of the network are the nodes without fanins. The *primary outputs* are the nodes without fanouts. The primary inputs and outputs define the external connections of the network.

The CNF formula of a combinational circuit is the conjunction of the CNF formulas of each gate, where the CNF formula of each gate denotes the valid input-output assignments to the gate. The derivation of CNF formulas of basic logic gates can be found in Larrabee [1992]. As a small example, consider the combinational circuit and its CNF formula given in Figure 4. In this Boolean formula, the first three clauses represent the CNF formula of the 2-input AND gate, and the last three clauses denote the CNF formula of the 2-input OR gate. Observe from Figure 4 that the assignment $x_1 = x_3 = x_4 = x_5 = 0$ and $x_2 = 1$ makes the formula φ equal to 1, indicating a valid assignment. However, the assignment $x_1 = x_3 = x_4 = 0$ and $x_2 = x_5 = 1$ makes the last clause of the formula equal to 0, and, consequently, the formula φ, indicating a conflict between the values of inputs and output of the OR gate.

2.3 0-1 Integer Linear Programming

The *0-1 Integer Linear Programming* (ILP) problem is the minimization or the maximization of a linear cost function, subject to a set of linear constraints, and is generally defined as follows:[1]

$$\text{Minimize} \quad \mathbf{w}^T \cdot \mathbf{x} \tag{1}$$
$$\text{Subject to} \ \ \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}, \quad \mathbf{x} \in \{0, 1\}^n \tag{2}$$

[1]The maximization objective can be easily converted to a minimization objective by negating the cost function. Less-than-or-equal and equality constraints are accommodated by the equivalences, $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b} \Leftrightarrow -\mathbf{A} \cdot \mathbf{x} \geq -\mathbf{b}$ and $\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \Leftrightarrow (\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}) \wedge (\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b})$, respectively.

In (1), w_j in $\mathbf{w}$ is an integer value associated with each of the n variables x_j, $1 \leq j \leq n$, in the cost function; and in (2), $\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}$ denotes the set of m linear constraints, where $\mathbf{b} \in \mathbb{Z}^m$, $\mathbf{w} \in \mathbb{Z}^n$, and $\mathbf{A} \in \mathbb{Z}^m \times \mathbb{Z}^n$. These linear constraints are commonly referred to as *pseudo-Boolean* (PB) constraints [Aloul et al. 2002].

A clause, $l_1 + \ldots + l_k$ where $k \leq n$, to be satisfied in a CNF formula can be interpreted as a linear inequality, $l_1 + \ldots + l_k \geq 1$, where $\overline{x_j}$ is represented by $1 - x_j$ as shown in Barth [1995]. These linear inequalities are the special cases of PB constraints, and are commonly referred to as CNF constraints, where $a_{ij} \in \{-1, 0, 1\}$ and b_i is equal to 1 minus the total number of complemented variables in its CNF formula. For instance, the set of clauses, $(x_1 + x_2)$, $(\overline{x_2} + x_3)$, and $(\overline{x_1} + \overline{x_3})$, has the equivalent linear inequalities given as $x_1 + x_2 \geq 1$, $-x_2 + x_3 \geq 0$, and $-x_1 - x_3 \geq -1$, respectively.

3. RELATED WORK

In this section we define the constant multiplication problems and summarize the previously proposed algorithms designed for these problems.

3.1 Multiple Constant Multiplications

The problem of finding the minimum number of operations solution for the Multiple Constant Multiplications (MCM) by a variable can be defined as follows:

Definition 3.1 *The MCM Problem*. Given a set of constants, $T = \{t_1, \ldots, t_k\} \subset \mathbb{N}$, composed of positive and odd unrepeated constants, find the minimum number of addition/subtraction operations that generate the constant multiplications.

The MCM problem has been proven to be an NP-complete problem in Cappello and Steiglitz [1984]. Note that the Single Constant Multiplication (SCM) problem is a special case of the MCM problem, when the number of elements of the set T is 1.

The straightforward way for the shift-adds realization of constant multiplications, generally known as the digit-based recoding method [Ercegovac and Lang 2003], initially defines the constants in multiplications in binary representation. Then, for each 1 in the binary representation of the constant, according to its bit position, it shifts the variable and adds up the shifted variables to obtain the result. For our example in Figure 2(a), the decompositions of constant multiplications $29x$ and $43x$ under binary are given as

$$29x = (11101)_{bin}x = x \ll 4 + x \ll 3 + x \ll 2 + x$$
$$43x = (101011)_{bin}x = x \ll 5 + x \ll 3 + x \ll 1 + x$$

where each constant multiplication requires 3 operations, (actually, the number of terms in its decomposed form minus 1), a total of 6 operations, as given in Figure 2(b). Furthermore, we can also define the constants under CSD representation, where each constant is represented with the minimum number of nonzero digits.

However, the digit-based recoding techniques do not consider the sharing of partial products among the constant multiplications that may yield an MCM design with a lesser number of operations. The algorithms that aim to maximize the partial product sharing in MCM can be categorized in two classes: the Common Subexpression Elimination (CSE) [Aksoy et al. 2008; Hartley 1996; Lefevre 2001; Park and Kang 2001] and graph-based (GB) methods [Aksoy et al. 2010; Dempster and Macleod 1995; Gustafsson et al. 2002; Voronenko and Püschel 2007]. Although the algorithms in both classes aim to find a solution with the fewest number of operations for the constant multiplications, by maximizing the partial product sharing, they differ in the search space that they explore. The CSE algorithms, which are also referred to as the pattern search methods, initially define the constants under a particular number representation, such as binary, CSD, or MSD. Then, all possible subexpressions are extracted from the representations of the constants and the "best" subexpression, generally the most common one, is chosen to be shared in constant multiplications. While the algorithm of Hartley [1996] iteratively extracts the most common 2-term subexpressions, the algorithm of Lefevre [2001] in each iteration finds a subexpression with the maximal number of terms and with at least 2 occurrences in the expressions. These heuristic algorithms generally define the constants under the CSD representation, where each constant has a unique representation with the minimum number of nonzero digits. The redundancy in the MSD representation of the constants is exploited in Park and Kang [2001], so that the proposed algorithm can consider alternative possible implementations of the constant multiplications. The exact CSE algorithm of Aksoy et al. [2008] that formalizes the MCM problem as a 0-1 ILP problem can handle the constants under binary, CSD, and MSD representations. For the example given in Figure 2(a), the exact CSE algorithm of Aksoy et al. [2008] finds the sharing of partial products $3x = (11)_{bin}x$ and $5x = (101)_{bin}x$ in both multiplications, when constants in multiplications $29x$ and $43x$ are defined in binary, and obtains a solution with 4 operations, as shown in Figure 2(c).

The GB algorithms are not restricted to any particular number representation while finding the common partial products, thus they obtain better solutions than CSE algorithms, as shown in Voronenko and Püschel [2007] and in Aksoy et al. [2010]. Prominent GB heuristics [Dempster and Macleod 1995; Voronenko and Püschel 2007] synthesize a constant multiplication using a single operation, including partial products in each iteration of their algorithms. While the partial product that can be implemented using the fewest number of operations is favored in Dempster and Macleod [1995], the partial product, which also has the maximum benefit over the implementation of the not-yet synthesized constant multiplications, is preferred in Voronenko and Püschel [2007]. The exact GB algorithms which find the minimum number of operations solution of the MCM problem using breadth-first and depth-first search were introduced in Aksoy et al. [2010]. For our example in Figure 2(a), the exact GB algorithm of Aksoy et al. [2010] finds a solution with the minimum number of 3 operations as $7x = x \ll 3 - x$, $29x = 7x \ll 2 + x$, and $43x = 7x \ll 1 + 29x$, sharing the common partial product $7x$. Observe that the partial product $7x = (111)_{bin}x$ cannot be extracted from the binary representations of both multiplications $29x$ and $43x$.

In addition to the CSE and GB algorithms, an intermediate approach was proposed in Wang and Roy [2005], where the partial products required for the realization of constant multiplications are explored using the differential coefficient technique [Muhammad and Roy 2002] which exploits the differences between the constant values. In this algorithm, the search space is represented by an undirected and complete graph, where the vertices denote the constants and the edges represent the differences between each vertex. The weight of each edge is defined as the implementation cost of the difference under a set of subexpressions, and is computed by the CSE algorithm of Hartley [1996]. Since different ways of subexpression elimination lead to different solutions, a genetic algorithm is applied to find a set of subexpressions, that yields the fewest number of operations in the implementation of constants based on their differences. Since the possible implementations of constant multiplications are extracted from the differences of absolute values of constants, it considers a smaller search space than that of a GB algorithm. However, it may find possible implementations that cannot be extracted by a CSE algorithm.

3.2 Constant Matrix-Vector Multiplication

The problem of multiplier-less realization of the constant matrix-vector multiplication (CMVM) operation using the fewest number of operations can be defined as follows:

Definition 3.2 *The CMVM Problem*. Given a constant matrix, $T_{k\times m}$, where $t_{ij} \in \mathbb{Z}$, $1 \leq i \leq k$, $1 \leq j \leq m$, to be multiplied by an $m \times 1$ input vector, find the minimum number of addition/subtraction operations that generate the linear transforms resulting from the multiplication of each row of the matrix $T_{k\times m}$ by the input vector.

Notice that the CMVM problem corresponds to an SCM problem when both k and m are 1, and to an MCM problem when k is greater than 1 and m is 1. Hence, the algorithms designed for the MCM problem can be extended to handle the CMVM problem instances. In this case, each constant in the set T and the single variable x in the MCM problem is replaced with a constant vector and an input vector, respectively. Thus, the CMVM problem is also an NP-complete problem due to the NP-completeness of the MCM problem.

The multiplier-less realization of linear transforms can also be obtained by the digit-based recoding method [Ercegovac and Lang 2003]. For our example given in Figure 3(a), the decomposed forms of linear transforms under binary are given as

$$\begin{aligned} y_1 &= 3x_1 + 11x_2 = (11)_{bin}x_1 + (1011)_{bin}x_2 \\ &= x_1 + 2x_1 + x_2 + 2x_2 + 8x_2 = x_1 + x_1 \ll 1 + x_2 + x_2 \ll 1 + x_2 \ll 3 \\ y_2 &= 5x_1 + 13x_2 = (101)_{bin}x_1 + (1101)_{bin}x_2 \\ &= x_1 + 4x_1 + x_2 + 4x_2 + 8x_2 = x_1 + x_1 \ll 2 + x_2 + x_2 \ll 2 + x_2 \ll 3 \end{aligned}$$

where the computation of y_1 and y_2 requires a total of 8 operations, as illustrated in Figure 3(b).

Furthermore, for the multiplierless realization of linear transforms, we can simply apply a constant multiplication algorithm on each constant of the matrix or on the constants of each column of the matrix. The sharing of constants in the rows of the matrix can be also utilized. Returning to our example in Figure 3(a), when the exact GB algorithm [Aksoy et al. 2010] is used as an MCM algorithm for the first column constants, the operations $3x_1 = x_1 \ll 1 + x_1$ and $5x_1 = x_1 \ll 2 + x_1$, and for the second column constants, the operations $3x_2 = x_2 \ll 1 + x_2$, $11x_2 = x_2 \ll 3 + 3x_2$, and $13x_2 = 3x_2 \ll 2 + x_2$ are obtained. Then, the linear transforms are implemented using two operations as $y_1 = 3x_1 + 11x_2$ and $y_2 = 5x_1 + 13x_2$, with a total of 7 operations. Similar to this approach, the algorithm of Potkonjak et al. [1996] uses a CSE technique that initially finds the common subexpressions for the columns, and then the common subexpressions for the rows.

The algorithms designed for the CMVM problem can again be categorized in two classes as CSE and GB algorithms. The CMVM problem was formalized as a 0-1 ILP problem in Yurdakul and Dündar [1999]. The possible implementations of linear transforms were found when constants are defined under a number representation in their decomposed forms. However, due to the exponential growth in the size of 0-1 ILP problems, the CSE algorithm [Yurdakul and Dündar 1999] only considers the 2-term subexpressions. The CSE heuristics [Arfaee et al. 2009; Boullis and Tisserand 2005; Hosangadi et al. 2005] initially determine each linear transform by simply multiplying each row of the constant matrix with the input vector, define the constants under CSD representation, and obtain the decomposed forms of linear transforms. Then, the sharing of common subexpressions is achieved based on heuristics. The algorithm of Hosangadi et al. [2005] selects the most common 2-term subexpression and eliminates its occurrences in the expressions in each iteration until there is no subexpression with a maximum number of occurrence greater than 1. For our example, given in Figure 3(a), the CSE heuristic [Hosangadi et al. 2005] first finds the subexpression $a = x_1 + x_2$ and then the subexpression $b = a + 8x_2$ and realizes the linear transforms with 4 operations, as given in Figure 3(c). The algorithm of Arfaee et al. [2009]

chooses its subexpressions based on a cost value which is computed as the product of the number of occurrences of a subexpression and its number of terms. The algorithm of Boullis and Tisserand [2005] relies on an efficient CSE algorithm [Lefevre 2001] that iteratively searches a subexpression with the maximal number of terms and with at least 2 occurrences. In Boullis and Tisserand [2005], the selection of a subexpression is also modified by taking into account the conflicts between the possible subexpressions.

The GB heuristic of Dempster et al. [2003] iteratively approaches the linear transforms to be implemented with the available expressions (in the beginning, the input variables $(x_1, x_2, \ldots, x_m)$ and their shifted values are available) by implementing subexpressions that require the minimum implementation cost as in the MCM algorithm of Dempster and Macleod [1995]. However, the GB algorithm of Dempster et al. [2003] is computationally intensive when compared to CSE algorithms, and can only be applied on small-size matrices. The method of Gustafsson et al. [2004] computes differences between each two rows of the matrix and determines the cost of each difference in terms of the total number of nonzero digits in CSD representation of each constant. Then, it finds a set of difference arrays with the minimum cost using a Minimum Spanning Tree (MST) algorithm, forms a new matrix with these arrays, where each row of the new matrix is one of the differences between two rows of the previous matrix, and continues the search with the new matrix. However, as stated in Gustafsson et al. [2004], this method is restricted in terms of the size and entries of the matrix due to the application of the MST algorithm in each iteration. Also, it cannot guarantee a solution with the minimum number of operations, although it finds the minimum cost difference arrays in each iteration.

4. THE EXACT CSE ALGORITHM

In this section, we describe the exact CSE algorithm designed for the multiplier-less realization of the linear transforms using the fewest numbers of operations. In the exact algorithm, finding the minimum number of operations solution is realized by finding the minimum number of subexpressions that are required for the implementation of the linear transforms and that require a single operation to be implemented. The exact algorithm consists of four main steps: (i) generation of all possible implementations of expressions; (ii) construction of the Boolean network; (iii) conversion of the Boolean network that represents the CMVM problem into a 0-1 ILP problem; and (iv) finding the minimum number of operations solution.

4.1 Generation of Subexpressions

In a preprocessing phase, each linear transform, also called a primary expression, is obtained by multiplying each row of the matrix with the input vector and is converted to an odd and positive expression, that is, the expression is divided by 2 until one of its constants is odd, and the expression is multiplied by -1, if the sign of the first nonzero constant in the expression is negative. Then, each primary expression is stored in a set called *Eset* without repetition. Thus, *Eset* includes all the necessary linear expressions to be synthesized.

The exact algorithm can handle the constants in the expressions under binary, CSD, and MSD representations. Initially, the constants in the expressions are defined under the given number representation and are decomposed such that each term in the expression is an element of the input vector, $x_1, x_2, \ldots, x_m$, multiplied by an integer power of two. Then, the implementations of primary expressions and subexpressions are found simply by partitioning the terms in the decomposed form of an expression in two parts using an addition operation. The part of the algorithm where the subexpressions are generated and determined for the implementation of expressions is given as follows.

(1) Take an unimplemented expression from *Eset*, $Eset_i$. Define the constants under a number representation, find the decomposed forms of the expression, and store all of its decomposition(s) in a set called $Dset_i$.[2] Form an empty set of arrays called $Sset_i$ associated with $Eset_i$. $Sset_i$ will contain all subexpressions that are required to implement $Eset_i$.
(2) Take a decomposition of $Eset_i$ from $Dset_i$, $Dset_i(j)$.[3]
(3) Find an operation from $Dset_i(j)$ that implements $Eset_i$;
   (a) Obtain the nonrepeated inputs of the operation. Make these subexpressions odd and positive. Store the subexpressions that are neither an element of the input vector nor a primary expression in an empty array called *Iarray*. Thus, *Iarray* may be empty, or it may contain one or two subexpressions.
   (b) If *Iarray* is empty, then make $Sset_i$ empty and go to Step 6. In this case, $Eset_i$ can be implemented using a single operation, whose inputs are input vector elements or primary expressions, which is the minimum implementation cost.
   (c) If *Iarray* is not empty, then check for each array of $Sset_i$, $Sset_i(k)$.[4]
      i. If $Sset_i(k)$ dominates[5] *Iarray*, then go to Step 4.
      ii. If *Iarray* dominates $Sset_i(k)$, then delete $Sset_i(k)$.
   (d) Add *Iarray* to $Sset_i$.
(4) Repeat Step 3 until all possible implementations of $Eset_i$ in $Dset_i(j)$ are considered.
(5) Repeat Step 2 until all decompositions of $Eset_i$ in $Dset_i$ are considered.
(6) Label $Eset_i$ as implemented. Add all subexpressions in $Sset_i$ to *Eset* without repetition and label them as unimplemented.
(7) Repeat Step 1 until all elements in *Eset* are labeled as implemented.

Note that *Eset* contains the primary expressions to be implemented in the beginning of the algorithm and it is augmented with the subexpressions required for the implementation of primary expressions in later iterations. Also, *Dset* includes the decomposed forms of primary expressions and subexpressions.

As an example, consider the linear transforms $y_1 = 5x_1 + 7x_2$ and $y_2 = x_1 + 3x_2$ to be synthesized. Table I presents the decomposed forms and all possible implementations of the linear transforms when constants are defined under CSD.

Observe from Table I that finding all possible implementations of an expression (Step 3 of the algorithm) is realized by partitioning its decomposed form in two subexpressions using an addition operation. For the primary expression $Eset_1$ whose decomposed form in CSD is $x_1 + 4x_1 - x_2 + 8x_2$ (with 4 terms), there exist 7 different implementations.[6] Since implementations $1-4$ include the input vector elements or their shifted versions (the first subexpressions of these implementations) as an input, the second subexpressions of these implementations are required as a single subexpression. Also, both subexpressions of implementations 5–7 are neither an input vector

[2]The number of decompositions in $Dset_i$, $|Dset_i|$, is equal to $\prod_{h=1}^{m} R(t_h)$, where m denotes the number of constants in the expression $Eset_i$ and $R(t_h)$ denotes the number of representations of a constant t_h under a given number representation, that is, binary, CSD, or MSD. Hence, under binary and CSD, $|Dset_i|$ is always equal to 1 and can be greater than 1 under MSD due to the alternative representations of a constant.
[3]j ranges from 1 to $|Dset_i|$.
[4]k ranges from 1 to the number of elements in $Sset_i$.
[5]A dominates B, if $A \bigcap B = A$.
[6]In general, the total number of implementations of an expression extracted from its decomposed form including n terms is $2^{n-1} - 1$.

Table I. All Possible Implementations of $5x_1 + 7x_2$ and $x_1 + 3x_2$ under CSD

$Eset = \{5x_1 + 7x_2, x_1 + 3x_2\}$
$Eset_1 = 5x_1 + 7x_2$
$Dset_1 = (101)x_1 + (100\overline{1})x_2 = x_1 + 4x_1 - x_2 + 8x_2$
Implementations of $5x_1 + 7x_2$:
1. $(x_1) + (4x_1 - x_2 + 8x_2)$, $Iarray = [4x_1 + 7x_2]$
2. $(4x_1) + (x_1 - x_2 + 8x_2)$, $Iarray = [x_1 + 7x_2]$
3. $(-x_2) + (x_1 + 4x_1 + 8x_2)$, $Iarray = [5x_1 + 8x_2]$
4. $(8x_2) + (x_1 + 4x_1 - x_2)$, $Iarray = [5x_1 - x_2]$
5. $(x_1 + 4x_1) + (-x_2 + 8x_2)$, $Iarray = [5x_1, 7x_2]$
6. $(x_1 - x_2) + (4x_1 + 8x_2)$, $Iarray = [x_1 - x_2, x_1 + 2x_2]$
7. $(x_1 + 8x_2) + (4x_1 - x_2)$, $Iarray = [x_1 + 8x_2, 4x_1 - x_2]$
$Sset_1 = \{[4x_1 + 7x_2], [x_1 + 7x_2], [5x_1 + 8x_2], [5x_1 - x_2], [5x_1, 7x_2], [x_1 - x_2, x_1 + 2x_2], [x_1 + 8x_2, 4x_1 - x_2]\}$
$Eset_2 = x_1 + 3x_2$
$Dset_2 = x_1 + (10\overline{1})x_2 = x_1 - x_2 + 4x_2$
Implementations of $x_1 + 3x_2$:
1. $(x_1) + (-x_2 + 4x_2)$, $Iarray = [3x_2]$
2. $(-x_2) + (x_1 + 4x_2)$, $Iarray = [x_1 + 4x_2]$
3. $(4x_2) + (x_1 - x_2)$, $Iarray = [x_1 - x_2]$
$Sset_2 = \{[3x_2], [x_1 + 4x_2], [x_1 - x_2]\}$

element nor a primary expression. Hence, these subexpressions are required as a pair for the implementation of $Eset_1$. For the primary expression $Eset_2$, since all its implementations include the input vector elements or their shifted versions (the first subexpressions of its implementations) as an input, the second subexpressions of these implementations are required as a single subexpression. After the subexpressions required for the implementation of an expression in *Eset* are determined, they are made odd and positive, are added to the *Eset*, and their implementations are found in a similar way.

Note that when constants in an expression are defined under binary or CSD, where a constant has a unique representation, there exists a single decomposition. However, the representation of the constants under MSD may allow possible representations of the constants increasing the number of decompositions and the possible sharing of subexpressions. For the expression $Eset_2$, $Dset_2$ includes two decompositions, $x_1 + x_2 + 2x_2$ and $x_1 - x_2 + 4x_2$, when constants are defined under MSD representation.

Also, in Step 3b of the algorithm, we determine that the implementation of an expression requires the minimum implementation cost, that is, a single operation, if there exists an operation implementing the expression with the input vector elements or primary expressions at its inputs. This is simply because all primary expressions will be implemented and the elements of the input vector are just inputs and require no implementation cost. Furthermore, in Step 3c of the algorithm, we avoid the repetition of a single or a pair of subexpressions required for the implementation of an expression and remove the redundant subexpressions determined by the dominance rule described in Coudert [1996].

4.2 Construction of the Boolean Network

After all subexpressions required to implement each primary expression and subexpression are found, these implementations are represented in a Boolean combinational

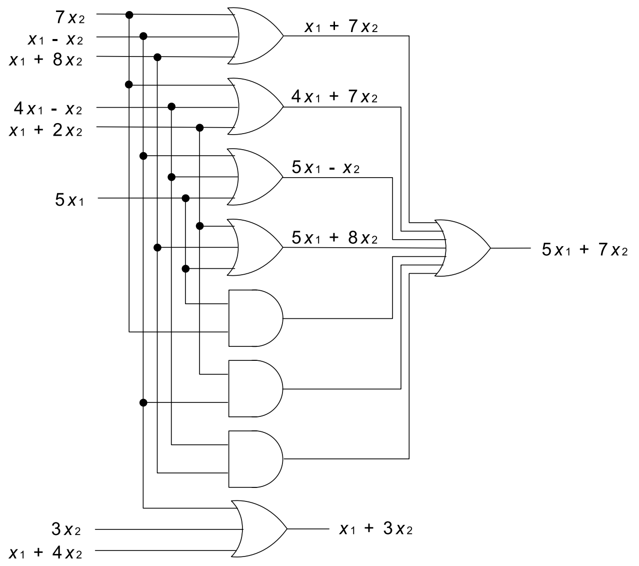


Fig. 5. The network generated for the linear transforms $5x_1 + 7x_2$ and $x_1 + 3x_2$ under CSD.

network that includes only AND and OR gates. The properties of the network are given as follows.

(1) The primary inputs of the network are the expressions that can be implemented using a single operation whose inputs are the elements of the input vector that the constants are multiplied with ($x_1, x_2, \ldots, x_m$), the primary expressions, or their shifted versions.
(2) An OR gate, representing a primary expression or a subexpression, combines all subexpressions that can be used for the implementation of the associated primary expression or subexpression.
(3) An AND gate represents a pair of subexpressions and combines two subexpressions.
(4) The outputs of the network are the OR gate outputs associated with the primary expressions.

The part of the algorithm where the Boolean network is constructed as follows.

(1) For each expression in *Eset*, *Eset$_i$*, if its associated *Sset*, *Sset$_i$*, is not empty, then generate an OR gate corresponding to the expression, otherwise make the expression as a primary input of the network.
(2) For each *Sset*, *Sset$_i$*, that is not empty, if it includes a pair of subexpressions, then generate an AND gate corresponding to this pair and assign its output to the input of the OR gate corresponding to the expression *Eset$_i$*. If it includes a single subexpression, that is, a primary input of the network or an OR gate output, then assign it to the input of the OR gate corresponding to the expression *Eset$_i$*.
(3) Identify all the OR gate outputs that represent a primary expression and set them as the outputs of the network.

The network generated for the linear transforms $5x_1 + 7x_2$ and $x_1 + 3x_2$ under CSD is presented in Figure 5. Observe that when the constants in an expression are defined under MSD, which introduces alternative representations of a constant compared to binary and CSD, all subexpressions required for the implementation of an expression will simply be the inputs of an OR gate representing the expression.

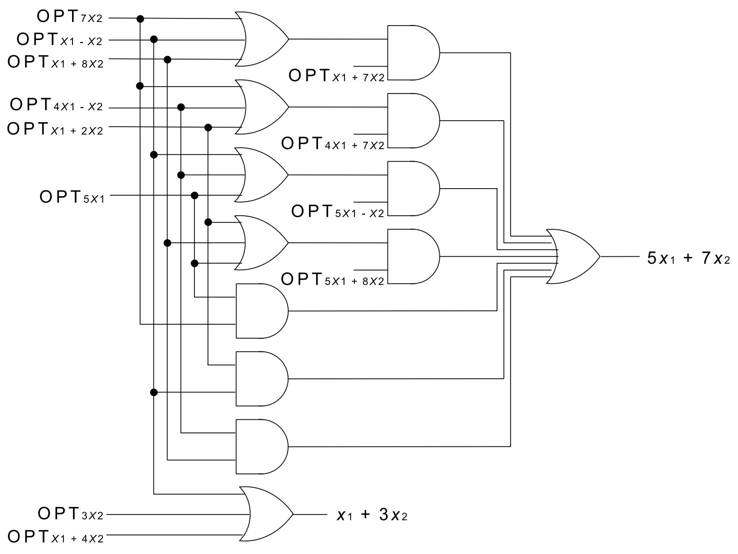


Fig. 6. Inclusion of the optimization variables to the network in Figure 5.

To construct the cost function to be minimized in the 0-1 ILP problem, we need to include optimization variables into the Boolean network. Since we aim to find the minimum number of operations solution of the CMVM problem by finding the minimum number of subexpressions, we associate the optimization variables with the subexpressions. Hence, for each subexpression that is represented as an output of an OR gate in the network, we add a 2-input AND gate, where one of its input is an optimization variable and the other is the output of the OR gate. For each subexpression that is a primary input of the network, we represent it as an optimization variable. The Boolean network after the optimization variables are included is presented in Figure 6.

4.3 Conversion to a 0-1 ILP Problem

After the Boolean network is generated, the conversion of the CMVM problem into a 0-1 ILP problem is straightforward, since the Boolean network in fact represents the optimization problem. Initially, the cost function of the 0-1 ILP problem is constructed as a linear function of the optimization variables, where the cost value of each optimization variable is 1. For our example given in Figure 6, the cost function is in the form of $OPT_{7x_2} + OPT_{x_1-x_2} + \ldots + OPT_{5x_1+8x_2}$. Then, the CNF formulas of each gate in the network are found and each clause in the CNF formulas is expressed as a linear inequality, as described in Section 2. Finally, the OR gate outputs that represent the primary expressions are assigned to 1, since our aim is to implement the primary expressions. Thus, the model obtained can serve as an input to a generic 0-1 ILP solver that will find the minimum cost solution (the minimum number of subexpressions), while respecting the constraints that represent the subexpressions that are required for the implementation of expressions.

4.4 Finding the Minimum Solution

After the minimum number of subexpressions required to implement the primary expressions are found, the primary expressions and subexpressions are synthesized in a bottom-up approach in order to realize the linear transforms in a reduced delay, that is, the maximum number of operations in series, generally known as the number of adder-steps. Hence, in the selection of an operation for the implementation of each primary expression and subexpression among possible operations whose inputs are in the found solution or the input vector elements, the one that leads to the smallest number of adder-steps is chosen.

4.5 Complexity of the Exact CSE Algorithm

As we cast the CMVM problem into a 0-1 ILP problem, the relevant complexity parameters are the number of variables, constraints, and optimization variables. In this section, we give the complexity analysis of the exact algorithm based on a single primary expression, assuming that the primary expression includes n terms, where $n \geq 3$, in its decomposed form when the constants are defined under a given number representation. Observe that n is directly related to the number of nonzero digits in the representation of the constants, which is also related to the bit-width of the constants, as shown in Faust and Chang [2010].

The number of subexpressions, $\#_{subexp}$, to be considered in an n term expression can be computed as $2^n - n - 2$. This value corresponds to the total number of all possible i element combinations on an n element set where i ranges from 0 to n, that is, 2^n, except when i is equal to 0, 1, and n. This comes from the fact that a subexpression extracted from an n term expression includes at least two terms and at most $n-1$ terms. Also, the number of primary inputs of the network, $\#_{pi}$, can be determined as $n(n-1)/2$, that is, the number of all possible 2-element combinations on an n element set. Note that in this case, a primary input of the network is a 2-term subexpression with the input vector elements or their shifted versions. Thus, the number of gates in the Boolean network is given by

$$\begin{aligned}
\#_{OR} &= \#_{subexp} - \#_{pi} + 1 \\
\#_{AND} &= (\#_{OR} - 1) + \sum_{i=4}^{n} \binom{n}{i} \sum_{j=2}^{i-2} \binom{i}{j} /2
\end{aligned}$$

where in the equation for $\#_{AND}$, while the first term of the addition operation, $(\#_{OR} - 1)$, stands for the number of AND gates including an optimization variable, the second term denotes the number of AND gates representing subexpression pairs. Note that a subexpression pair is found in the implementation of an expression including at least 4 terms (corresponding to the initial value of i of the first sum in the second term of $\#_{AND}$ equation), which can easily be observed on the example in Table I. Also, observe that in a subexpression pair, each subexpression has at least 2 terms (corresponding to the initial value of j of the second sum in the second term of $\#_{AND}$ equation), and due to the distributive law of addition, there exist repeated subexpression pairs that are avoided by dividing 2 in the equation of $\#_{AND}$. As an example, consider the expression $5x_1 + 7x_2$ that includes $n = 4$ terms when the constants are defined in CSD, as shown in Table I. The network constructed for this expression includes 7 AND and 5 OR gates, as can be seen in the related part of Figure 6.

Thus, the number of variables, constraints, and optimization variables in the 0-1 ILP problem can be given as follows:

$$\begin{aligned}
\#_{vars} &= \#_{subexp} + \#_{OR} + \#_{AND} \\
\#_{cons} &= 3(\#_{AND}) + \sum_{i=3}^{n} \binom{n}{i} 2^{i-1} \\
\#_{optvars} &= \#_{subexp}
\end{aligned}$$

Table II. Upper Bounds on the Size of Boolean Network and 0-1 ILP Problem

n	$\#_{AND}$	$\#_{OR}$	$\#_{vars}$	$\#_{cons}$	$\#_{optvars}$
4	7	5	22	45	10
6	171	42	269	841	56
8	2,255	219	2,720	9,981	246
10	24,403	968	26,383	102,633	1,012
12	241,143	4,017	249,242	989,005	4,082
14	2,276,795	16,278	2,309,441	9,221,673	16,368
16	20,999,071	65,399	21,129,988	84,520,317	65,518

where in the equation for $\#_{cons}$, the first term denotes the number of constraints obtained from the AND gates[7] in the network and the second term stands for the number of constraints obtained from the OR gates in the network.

Table II gives the size of a Boolean network in terms of the number of AND and OR gates, and the size of 0-1 ILP problem in terms of the number of variables, constraints, and optimization variables for different values of n.

4.6 Limitations of the Exact CSE Algorithm

As can be easily observed from Table II, the size of 0-1 ILP problem grows exponentially with the number of terms in the decomposed form of the expression denoted by n. We note that current SAT-based 0-1 ILP solvers can handle the problems that are generated for an expression including up to 12 terms. Thus, the limitations of the exact CSE algorithm on the constant matrices, in terms of the size of the matrix and the bit-widths of the constants in the matrix, can be derived. Assume that the constants in the matrix are defined under binary. Thus, the exact algorithm can be applied up to $k \times 12$ matrices including only 1, 0, and -1, that is, 1-bit constants, where k denotes both the number of rows of the matrix and the number of linear transforms. It can also be applied up to $k \times 6$ and $k \times 4$ matrices when the constants in these matrices are 2-bit and 3-bit constants, respectively. However, by representing the constants under CSD (with the minimum number of nonzero digits), the exact algorithm can be applied on larger size matrices including larger constants. Also, the limitations of the exact CSE algorithm on the number of rows of the constant matrix depend heavily on the constants in the linear transforms. Although it may increase the size of the 0-1 ILP problem by considering alternative subexpressions that are not encountered in other linear transforms, it may not add too much complexity if most of the subexpressions required for a linear transform are shared among other linear transforms.

In the exact CSE algorithm, all possible implementations of an expression are extracted when the constants in the expression are defined under a number representation. Thus, the number of operations in its minimum solution depends on the number representation. Returning to our example in this section, for the linear transforms $y_1 = 5x_1+7x_2$ and $y_2 = x_1+3x_2$ under CSD, the minimum solution includes 4 operations, $a = x_1 - x_2$, $b = a + x_2 \ll 3$, $y_1 = b + x_1 \ll 2$, and $y_2 = a + x_2 \ll 2$. When the constants are defined under binary, the minimum solution consists of 3 operations, $c = x_1 + x_2$, $y_2 = c + x_2 \ll 1$, and $y_1 = y_2 + c \ll 2$. Moreover, the exact CSE algorithm may not consider the implementations of an expression taken into account in an algorithm which is not restricted to any particular number representation. As a simple example, consider the linear transforms, $y_1 = x_1 + x_2$, $y_2 = x_1 + x_3$, and $y_3 = 2x_1 + x_2 + x_3$. The minimum

[7]Every AND gate in the Boolean network has 2 inputs.

solution consists of 3 operations,[8] $y_1 = x_1 + x_2$, $y_2 = x_1 + x_3$, and $y_3 = y_1 + y_2$. However, the solution of the exact CSE algorithm under binary, CSD, and MSD representation includes 4 operations, $y_1 = x_1 + x_2$, $y_2 = x_1 + x_3$, $d = x_2 + x_3$, and $y_3 = d + x_1 \ll 1$. Thus, the solution of the exact CSE algorithm is in fact a local minimum solution of the CMVM problem, since the search space that the exact CSE algorithm covers is the subspace of an exact GB algorithm that is not restricted by any particular number representation. Yet, to the best of our knowledge, no exact GB algorithm has been proposed for the CMVM problem.

However, the local minimum solution of the exact CSE algorithm can be guaranteed as the global minimum on special instances, such as the constant matrices that consist of only 1, 0, and -1 (the error-correcting codes [Potkonjak et al. 1996] are generally in this form). Its local minimum solution can also be ensured as the global minimum if it is equal to the lower bound on the number of operations in a CMVM problem [Gustafsson 2007]. We note that the solution of the exact CSE algorithm on linear transforms given in Figure 3(a), (the same as in Figure 3(c)) and its solution on linear transforms $5x_1 + 7x_2$ and $x_1 + 3x_2$ when constants are defined under binary are the global minimum solutions determined with the use of lower bound given in Gustafsson [2007].

5. THE APPROXIMATE ALGORITHMS

As described in Sections 4.5 and 4.6, there are still instances that the exact CSE algorithm cannot cope with. Hence, as for all NP-complete problems, a heuristic algorithm that can find a solution close to the minimum using little computational effort is always desirable. In this section, we introduce a CSE heuristic algorithm, called H_{2MC}, based on the CSE heuristics [Hartley 1996; Hosangadi et al. 2005] that iteratively utilize the most common (MC) 2-term subexpressions. However, the subexpression selection heuristic, which significantly affects the final solution due to the iterative decision making, can be improved by choosing an MC 2-term subexpression, among many others, in an iteration such that its selection leads to the least loss of subexpression sharing in the next iterations. These subexpressions are called the most common minimum conflicting (MCmc) 2-term subexpressions. Since the CSE algorithms in this scheme are restricted to a particular number representation, we also present a hybrid algorithm, called HCMVM, which iteratively finds promising realizations of linear transforms using a numerical difference method and applies H_{2MC} to achieve the sharing of common subexpressions.

5.1 Implementation of the H_{2MC} Algorithm

The H_{2MC} algorithm can handle the constants under binary and CSD representations where there is a unique representation for each constant. As in the exact CSE algorithm, in the preprocessing phase of H_{2MC}, each primary expression is obtained by multiplying each row of the matrix with the input vector, converted to an odd and positive expression, and stored in a set called *Eset* without repetition. The decompositions of expressions when the constants are defined under a given number representation are found and stored in a set called *Dset*. Also, an empty set called *Sset*, which will store the 2-term subexpressions selected in each iteration is formed. The part of the

[8]Note that k distinct linear transforms cannot be implemented using less than k operations, as k constant multiplications cannot be realized using less than k operations, as proved in Dempster and Macleod [1995].

Table III. Procedure of H_{2MC} on Linear Transforms $15x_1 + 43x_2$ and $38x_1 + 51x_2$ under CSD

Initial expressions:	Initial expressions in *Dset*:
$y_1 = 15x_1 + 43x_2$	$Dset_1 = -x_1 + 16x_1 - x_2 - 4x_2 - 16x_2 + 64x_2$
$y_2 = 38x_1 + 51x_2$	$Dset_2 = -2x_1 + 8x_1 + 32x_1 - x_2 + 4x_2 - 16x_2 + 64x_2$
Iteration 1	
MC 2-terms: $x_2 + 16x_2$, $-x_2 + 4x_2$ (#occurrences = 3)	
MCmc 2-terms: $x_2 + 16x_2$, $-x_2 + 4x_2$	
Current expressions in *Dset*:	Current expressions in *Sset*:
$Dset_1 = -x_1 + 16x_1 - 4x_2 + 64x_2 - a$	$Sset_1 = a = x_2 + 16x_2$
$Dset_2 = -2x_1 + 8x_1 + 32x_1 - a + 4a$	
Iteration 2	
MC 2-terms: $-x_1 + 16x_1$, $x_1 + 4x_2$, $2x_1 + a$ (#occurrences = 2)	
MCmc 2-terms: $x_1 + 4x_2$, $2x_1 + a$	
Current expressions in *Dset*:	Current expressions in *Sset*:
$Dset_1 = -x_1 + 16x_1 - 4x_2 + 64x_2 - a$	$Sset_1 = a = x_2 + 16x_2$
$Dset_2 = 32x_1 - b + 4b$	$Sset_2 = b = 2x_1 + a$
Iteration 3	
MC 2-terms: $x_1 + 4x_2$ (#occurrences = 2)	
MCmc 2-terms: $x_1 + 4x_2$	
Current expressions in *Dset*:	Current expressions in *Sset*:
$Dset_1 = -a - c + 4c$	$Sset_1 = a = x_2 + 16x_2$
$Dset_2 = 32x_1 - b + 4b$	$Sset_2 = b = 2x_1 + a$
	$Sset_3 = c = x_1 + 4x_2$

algorithm, where the MCmc 2-term subexpressions are found and replaced in the decompositions of expressions, follows.

(1) For each 2-term subexpression that is extracted from the decompositions of expressions in *Dset*, convert the subexpression to odd and positive, find the occurrences of the subexpression in the elements of *Dset* considering its shifted and negated versions, and determine the MC 2-term subexpressions.
(2) If the number of occurrence of the MC 2-term subexpressions in *Dset* is 1, then return *Dset* and *Sset*.
(3) Otherwise, find the minimum conflicting 2-term subexpressions in the MC 2-term subexpressions, that is, the MCmc 2-term subexpressions.
(4) Choose one of the MCmc 2-term subexpressions, add it to *Sset* by labeling it with a variable, replace its occurrences in *Dset* with its label, and go to Step 1.

We note that the number of 2-term subexpressions to be considered in a decomposed form of an expression including n terms is $n(n-1)/2$. Thus, the maximum number of 2-term subexpressions considered in Step 1 of the algorithm is simply $\sum_{i=1}^{k} n_i(n_i - 1)/2$, where k and n_i denote the number of expressions and the number of terms in the i^{th} decomposed form of the expression, respectively.

The procedure of H_{2MC} is described via the example of Table III using the linear transforms $y_1 = 15x_1 + 43x_2$ and $y_2 = 38x_1 + 51x_2$. Suppose that the constants are defined under CSD. In the first iteration, two MC 2-term subexpressions that occur both in y_1 once and in y_2 twice, with a total of 3 occurrences, are obtained. Note that the occurrences can be also found in shifted or negated forms. However, the occurrences of the MC 2-term subexpressions in the linear transforms conflict with each other, indicating that selecting one of them will eliminate the other in the next iteration. Hence, the MCmc 2-term subexpressions are determined as MC 2-term subexpressions. In this iteration, the subexpression $x_2 + 16x_2$ is chosen and replaced in the expressions. In the second iteration, there are three MC 2-term subexpressions with two occurrences. The subexpressions $x_1 + 4x_2$ and $2x_1 + a$ occur only in y_1 and y_2, respectively. The subexpression $-x_1 + 16x_1$ occurs in both expressions and its occurrences conflict with the occurrences of both $x_1 + 4x_2$ and $2x_1 + a$. Thus, the MCmc 2-term subexpressions are determined as $x_1 + 4x_2$ and $2x_1 + a$. In this iteration, $2x_1 + a$ is chosen and replaced in the expressions. Hence, in the third iteration, the $x_1 + 4x_2$ is encountered again, is selected, and replaced in the expressions. The resulting expressions do not include 2-term subexpressions with a maximum number of occurrence greater than 1. Thus, the solution is obtained with the 2-term subexpressions selected in each iteration (the elements of the final *Sset*) and the elements of the final *Dset*, with a total of 7 operations, 3 for the chosen subexpressions and 4 for the final expressions.

To illustrate the effect of selecting an MCmc 2-term subexpression, observe that if the MC 2-term subexpression $-x_1 + 16x_1$ was selected in the second iteration, there would not be any 2-term with a maximum occurrence greater than 1 in the next iteration, since this subexpression would remove the occurrences of $x_1 + 4x_2$ and $2x_1 + a$. Thus, 8 operations would be required to implement the linear transforms.

Furthermore, as stated in Aksoy et al. [2007], finding the fewest number of operations does not always lead to a design with optimal area at gate-level. Hence, to further reduce the area of a CMVM design, while synthesizing the expressions in the final *Dset*, we apply some hardware optimizations that do not change the number of operations but change their realizations. Thus, for each expression in the final *Dset* including more than 2 terms, we initially separate the terms into two sets *Pset* and *Mset*, considering their sign. This comes from the fact that although the cost of an adder and a subtracter is assumed to be equal in high-level algorithms, a subtracter occupies a larger area than an adder at gate-level. Then, in each set, we iteratively select two terms that have the smallest bit-width, that is, the narrowest, to be realized using an adder in order to reduce the size of the operation. Finally, if *Mset* is not empty, we use a subtracter to realize the expression. Thus, for our example in Table III, $Dset_1$ and $Dset_2$ in the final *Dset* are implemented as $4c - (a + c)$ and $(32x_1 + 4b) - b$, respectively.

5.2 Implementation of the HCMVM Algorithm

Although CSE algorithms are computationally efficient due to their greedy heuristics, their solution depends heavily on the number representation used in defining the constants. On the other hand, using the difference method described in Muhammad and Roy [2002] and Wang and Roy [2005], promising realizations of linear transforms can be obtained by exploring the differences of linear transforms. Since all linear transforms are to be implemented at the end, a difference that requires fewer operations can be chosen to implement a linear transform that requires more operations. Furthermore, as a set of promising differences is found for the implementation of linear transforms, H_{2MC} can be applied to find common subexpressions to be shared among these expressions.

In the preprocessing phase of HCMVM, each linear transform is converted to an odd and positive expression and is stored in a set called *Eset* without repetition. Then, as done in the GB algorithms designed for the MCM problem [Aksoy et al. 2010; Dempster and Macleod 1995; Voronenko and Püschel 2007], the linear transforms that can be synthesized using a single operation whose inputs are an element of the input vector, an implemented linear transform, or their shifted versions are found iteratively and moved from *Eset* to *Iset* which will include the implemented expressions. As a simple example, consider the linear transforms $y_1 = x_1 + 4x_2$, $y_2 = 2x_1 + x_3$, and $y_3 = 5x_1 + 4x_2 + 2x_3$. After the linear transforms y_1 and y_2 are implemented using a single operation with the input variables, y_3 can be synthesized as $y_3 = y_1 + y_2 \ll 1$. Hence, this is the optimal part, meaning that when all the linear transforms are realized in this part, the minimum number of operations solution is obtained.

If there are still linear transforms in *Eset* after the optimal part, the algorithm switches to its heuristic part. In this part, it initially finds a solution on expressions in *Eset* with the CSE algorithm, H_{2MC}, and records its solution as the best solution found so far. Then, the cost of each linear transform in *Eset* is computed as the total number of nonzero digits of each constant in a number representation, binary or CSD, which H_{2MC} uses in the definition of constants. The linear transforms are sorted in a descending order based on their cost values. For each expression in *Eset*, $Eset_i$, with its cost value $cost_i$, where $i < k$ and k denotes the number of expressions in *Eset*, all the differences of $Eset_i$ with an expression in *Eset*, $Eset_j$, where $i < j \leq k$, are computed as $d_{ij} \ll l_1 = Eset_i - Eset_j \ll l_2$, where $l_1, l_2 \geq 0$ denote the left shifts. The cost of each difference is determined in terms of the total number of nonzero digits of each constant under the given number representation and a difference with the minimum cost value $cost_d$ is determined. If $cost_d < cost_i - 1$, then $Eset_i$ is moved from *Eset* to *Iset*, and the difference with the minimum cost is added into *Eset* in place of $Eset_i$. After all differences for each expression in *Eset* except $Eset_k$ are explored, H_{2MC} is applied on the expressions in *Eset* and a set of operations realizing the expressions in *Eset* is obtained. If its solution considering the elements in *Iset* is better than the best one that has been found so far, it is updated with this solution. HCMVM iterates until there are no more differences that can be replaced with the expressions in *Eset*.

The procedure of HCMVM is illustrated on Example 1 of Gustafsson et al. [2004] when H_{2MC} defines the constants under CSD as given in Table IV. In this table, the values between parenthesis next to the expressions denote the respective cost values. Initially, H_{2MC} is applied on linear transforms and a solution with 19 operations is obtained. Then, in the first iteration of HCMVM, the linear transforms $Eset_1$, $Eset_2$, and $Eset_3$ are realized using a single operation whose inputs are an element of *Eset* and a difference with the minimum cost. They are synthesized as $Eset_1 = Eset_2 + d_{12}$, $Eset_2 = Eset_3 + d_{23}$, $Eset_3 = Eset_4 + d_{34} \ll 1$. Then, these linear transforms are moved from *Eset* to *Iset* and the associated differences are added to *Eset*. In this case, H_{2MC} finds a solution with 10 operations on *Eset*. Thus, a total of 13 operations are required, considering that the expressions in *Iset* are synthesized using a single operation. In the second iteration, HCMVM follows the same procedure realizing $Eset_1$ as $Eset_4 + d_{14} \ll 1$ and finding a solution with a total of 13 operations again. Since there are no more promising differences, HCMVM takes only two iterations. As reported in Gustafsson et al. [2004], the algorithms of Dempster et al. [2003] and Gustafsson et al. [2004] find a solution with 14 operations on this instance.

Thus, the HCMVM algorithm may find better solutions than CSE algorithms when a linear transform including a large number of terms in its decomposed form can be implemented using a single operation that requires an expression found by the difference method, which includes a few terms in its decomposed form.

6. EXPERIMENTAL RESULTS

In this section, we compare the exact and approximate algorithms with previously proposed algorithms on randomly generated constant matrices and linear DSP transforms.[9] This section is divided in three parts. In the first part, the instances which

[9]The algorithms and instances are available at http://algos.inesc-id.pt/multicon.

Table IV. Procedure of HCMVM on Example 1 of Gustafsson et al. [2004]

Initial expressions:	
$y_1 = 7x_1 + 8x_2 + 2x_3 + 13x_4$	
$y_2 = 12x_1 + 11x_2 + 7x_3 + 13x_4$	
$y_3 = 5x_1 + 8x_2 + 2x_3 + 15x_4$	
$y_4 = 7x_1 + 11x_2 + 7x_3 + 11x_4$	
Solution of H_{2MC} on initial expressions: 19 operations	
Iteration 1	
The expressions of *Eset* and chosen differences:	
$Eset_1(10) : 12x_1 + 11x_2 + 7x_3 + 13x_4$	$d_{12}(3) : 5x_1 + 2x_4$
$Eset_2(10) : 7x_1 + 11x_2 + 7x_3 + 11x_4$	$d_{23}(5) : 3x_2 + 5x_3 - 2x_4$
$Eset_3(7) : 7x_1 + 8x_2 + 2x_3 + 13x_4$	$d_{34}(2) : x_1 - x_4$
$Eset_4(6) : 5x_1 + 8x_2 + 2x_3 + 15x_4$	
Current expressions in *Eset*:	Current expressions in *Iset*:
$5x_1 + 2x_4$	$12x_1 + 11x_2 + 7x_3 + 13x_4$
$3x_2 + 5x_3 - 2x_4$	$7x_1 + 11x_2 + 7x_3 + 11x_4$
$x_1 - x_4$	$7x_1 + 8x_2 + 2x_3 + 13x_4$
$5x_1 + 8x_2 + 2x_3 + 15x_4$	
Solution of H_{2MC} on *Eset*: 10 operations	Total: 10 + 3 = 13 operations
Iteration 2	
The expressions of *Eset* and chosen differences:	
$Eset_1(6) : 5x_1 + 8x_2 + 2x_3 + 15x_4$	$d_{14}(4) : 2x_1 + 4x_2 + x_3 + 8x_4$
$Eset_2(5) : 3x_2 + 5x_3 - 2x_4$	
$Eset_3(3) : 5x_1 + 2x_4$	
$Eset_4(2) : x_1 - x_4$	
Current expressions in *Eset*:	Current expressions in *Iset*:
$2x_1 + 4x_2 + x_3 + 8x_4$	$12x_1 + 11x_2 + 7x_3 + 13x_4$
$3x_2 + 5x_3 - 2x_4$	$7x_1 + 11x_2 + 7x_3 + 11x_4$
$5x_1 + 2x_4$	$7x_1 + 8x_2 + 2x_3 + 13x_4$
$x_1 - x_4$	$5x_1 + 8x_2 + 2x_3 + 15x_4$
Solution of H_{2MC} on *Eset*: 9 operations	Total: 9 + 4 = 13 operations

the exact CSE algorithm can handle were used to compare the results of CSE heuristics with the minimum solutions and to determine the limitations of the exact CSE algorithm. Then, large-size problem instances, which go beyond the applicability of the exact CSE algorithm, were used to compare the CSE heuristics. Finally, the gate-level results of DCT designs, which were obtained from the direct realization of linear transforms and from the solutions of the CSE heuristic [Hosangadi et al. 2005] and HCMVM algorithms, are presented.

6.1 Comparison of Exact and Heuristic Algorithms

As the first experiment set, we used randomly generated $k \times 2$ constant matrices, where k is 2, 5, 10, 15, and 20. The constants are generated in the range $[-2^6 + 1, 2^6 - 1]$ with 40 instances for each type of matrix. Table V presents the results of various algorithms in terms of the average number of operations: the exact GB MCM algorithm [Aksoy et al. 2010] when it is applied on the constants of each column of the constant matrix; the heuristic of [Hosangadi et al. 2005], H_{2MC}, and HCMVM when constants are defined under CSD; and the exact CSE algorithm under CSD and MSD.

Table V. Summary of Results of the Algorithms on Randomly Generated $k \times 2$ Matrices

Algorithms	2×2	5×2	10×2	15×2	20×2
[Aksoy et al. 2010]	6.2	13.7	25.2	35.6	45.0
[Hosangadi et al. 2005] - CSD	6.1	13.7	23.5	32.8	41.7
H_{2MC} - CSD	6.0	13.6	23.4	32.7	41.4
HCMVM - CSD	5.8	12.3	21.1	29.1	36.7
Exact CSE - CSD	6.0	13.3	22.3	30.9	38.5
Exact CSE - MSD	5.4	12.1	20.6	28.6	35.8

Table VI. Size of 0-1 ILP Problems on Randomly Generated $k \times k$ Matrices under CSD

Ins.	Type	vars	cons	optvars
1	2×2	2592	9487	247
2	2×2	4982	18413	448
3	4×4	7018	25195	701
4	4×4	7034	25252	696
5	6×6	8841	31995	818
6	6×6	12883	47668	988
7	8×8	1832	6061	277
8	8×8	2114	7057	315

Observe from Table V that although the minimum number of operations solution on the constants of each column of a matrix is obtained by the exact GB algorithm [Aksoy et al. 2010], since the sharing of partial products is limited within the constants in each column, it obtains worse solutions than algorithms designed specifically for the CMVM problem. The results of the CSE heuristic [Hosangadi et al. 2005] and H_{2MC} are close to each other, and also to the minimum solutions obtained by the exact CSE algorithm under CSD on instances, where k is 2, 5, and 10. On the other hand, HCMVM obtains significantly better solutions than these CSE heuristics, where the difference of the average number of operations between these heuristics and HCMVM is almost 5 operations on 20×2 matrices. Also, for each matrix type, on average, HCMVM obtains better solutions than those of the exact CSE algorithm when constants are defined under CSD, since HCMVM considers alternative implementations of linear transforms obtained with the difference method which may not be considered in the exact CSE algorithm. However, the use of the MSD representation in the exact CSE algorithm increases the number of possible implementations of linear transforms, yielding better solutions than all the algorithms given in Table V.

As the second experiment set, we used 8 randomly generated $k \times k$ matrices, where k is 2, 4, 6, and 8, and constants were generated in between $\left[-2^8+1, 2^8-1\right]$, $\left[-2^4+1, 2^4-1\right]$, $\left[-2^2+1, 2^2-1\right]$, and $[-1, 1]$, respectively. The size of 0-1 ILP problems generated by the exact CSE algorithm under CSD is given in Table VI, where *vars*, *cons*, and *optvars* denote the number of variables, constraints, and optimization variables, respectively.

Table VII presents the results of algorithms on randomly generated $k \times k$ matrices. In this table, *op* and *as* denote the number of operations and adder-steps (maximum number of operations in series), respectively. While *CPU* stands for the CPU time in seconds for the heuristic algorithms, all written in MATLAB, it represents the CPU time of the SAT-based 0-1 ILP solver *minisat+* used to find the minimum solution [Een

Table VII. Summary of Results of the Algorithms on Randomly Generated $k \times k$ Matrices

Ins.	[Hosangadi et al. 2005]			H_{2MC}			HCMVM			Exact CSE - CSD		
	op	as	CPU	op	as	CPU	op	as	CPU	op	as	CPU
1	8	4	0.1	7	4	0.0	7	4	0.2	7	5	6.3
2	10	4	0.1	10	4	0.0	8	4	0.0	9	4	128.0
3	19	4	0.1	19	4	0.1	19	4	0.1	18	5	33053.7
4	18	4	0.1	18	4	0.1	15	4	0.1	18	5	7733.2
5	26	4	0.1	25	4	0.2	22	5	0.1	24	7	30295.4
6	26	4	0.2	26	4	0.2	24	5	0.2	24	5	11581.6
7	22	4	0.1	22	4	0.2	20	4	0.1	20	5	28.1
8	21	4	0.1	21	4	0.1	21	4	0.1	19	5	21.2
Total	150	32	0.9	148	32	0.9	136	34	0.9	139	41	82847.5

and Sorensson 2006]. Note that all the algorithms were run on a PC with Intel Xeon at 3.16GHz and 8GB memory, and their solutions were obtained when the constants are defined under the CSD representation.

Observe from Table VII that although the exact CSE algorithm finds similar and better solutions than the CSE heuristic [Hosangadi et al. 2005] and H_{2MC}, obtaining the minimum solution may require much more computational effort with respect to that required for the heuristic algorithms, restricting its application only to small-size instances. Observe from Tables VI and VII that the required computation time for the exact CSE algorithm depends on the size of 0-1 ILP problem which in turn depends heavily on the constants in the matrices. Also, HCMVM may find better solutions than the exact CSE algorithm, since promising realizations of linear transforms can be found using a numerical difference method.

These experiments indicate that the exact CSE algorithm can only be applied on small-size instances due to the NP-completeness of the problem. However, there are instances where the exact CSE algorithm finds better solutions than the CSE heuristics in a reasonable time. Also, the use of MSD representation yields better solutions than those of the CSE algorithms that define the constants under CSD, since possible alternative implementations of linear transforms increase the sharing of subexpressions. The CSE heuristics can find solutions close to the minimum guaranteed by the exact CSE algorithm using little computational effort. The hybridization of CSE and numerical methods in HCMVM leads to significant reductions in the number of operations when compared to the CSE algorithms.

6.2 Comparison of Heuristic Algorithms

As the third experiment set, we used $k \times k$ matrices where k varies in between 2 and 16, consisting of randomly generated 8-bit constants, with 100 instances for each matrix type. Table VIII presents the results of the CSE heuristics when constants are defined under CSD. In this table, *stdev* denotes the standard deviation. Note that the results of the algorithm [Boullis and Tisserand 2005] were taken from its paper.

As can be observed from Table VIII, the heuristic algorithms [Hosangadi et al. 2005; Boullis and Tisserand 2005], and H_{2MC} obtain similar solutions on $k \times k$ matrix instances, where k is up to 6. For instances, where k is greater than 6, H_{2MC} obtains better solutions than the heuristic of Hosangadi et al. [2005], where the difference in the average number of operations between these algorithms reaches up to 9.76 on 16×16 matrices. This is simply because while the heuristic of Hosangadi et al. [2005] iteratively extracts the MC 2-term subexpressions, H_{2MC} finds the most promising one among these MC 2-term subexpressions, that is, the minimum conflicting terms.

Table VIII. Summary of Results of the Algorithms on $k \times k$ Randomly Generated Matrices with 8-bit Constants

k	[Hosangadi et al. 2005]		[Boullis and Tisserand 2005]		H_{2MC}		HCMVM	
	avg.	stdev	avg.	stdev	avg.	stdev	avg.	stdev
2	8.79	1.12	9.7	1.3	8.73	1.04	8.16	0.93
3	18.28	1.66	17.1	0.9	18.07	1.60	16.47	1.11
4	32.15	2.16	31.2	2.2	31.70	2.04	27.62	1.61
5	48.85	2.64	47.1	3.3	48.16	2.42	41.38	1.82
6	67.81	2.81	66.1	4.0	66.52	2.81	57.29	2.18
7	90.30	3.49	88.9	5.3	88.77	3.26	75.42	2.36
8	116.26	3.63	113.2	6.7	114.10	3.49	96.27	2.80
9	144.41	4.02	141.6	7.0	141.94	4.16	119.09	3.03
10	175.67	5.22	172.4	8.5	172.03	4.60	143.49	3.21
11	209.04	4.72	207.1	10.8	205.04	4.32	170.59	3.36
12	246.30	4.79	241.6	11.9	240.87	4.14	200.39	3.71
13	284.89	5.64	279.6	13.3	278.61	5.12	231.40	3.71
14	326.60	6.55	322.9	17.0	320.00	5.33	264.26	4.39
15	370.05	5.60	370.0	20.0	360.94	5.61	300.44	4.09
16	417.25	6.50	412.4	19.4	407.49	6.60	338.33	4.89

Table IX. Summary of Results of the Algorithms on Linear DSP Transforms

Algorithms	H.264	DCT8	IDCT8	DHT	DST
[Potkonjak et al. 1996]	–	227	222	211	252
[Nguyen and Chatterjee 2000]	–	202	183	209	238
[Arfaee et al. 2009]	53	161	140	161	181
[Hosangadi et al. 2005] - CSD	51	147	138	159	176
H_{2MC} - CSD	49	150	137	150	174
HCMVM - CSD	42	145	136	150	172

Besides, the heuristic of Boullis and Tisserand [2005], which considers the subexpressions with the maximal number of terms and with at least 2 occurrences, obtains similar solutions to H_{2MC}. On the other hand, HCMVM finds the best solutions among the algorithms where the maximum difference on the average number of operations between H_{2MC} and HCMVM is 69.16, obtained on 16×16 matrices. Also, observe that the standard deviation values on the results of HCMVM are smaller than those of the CSE heuristics, indicating that its solutions are close to the average value.

The fourth experiment set was provided by Kastner, and consists of a 7×3 H.264 video compression transform, an 8-point DCT, an 8-point Inverse DCT (IDCT), an 8×8 Discrete Hartley Transform (DHT), and an 8×8 Discrete Sine Transform (DST), where the constants are defined under 14 bits. The solutions of the high-level algorithms in terms of the number of operations are given in Table IX, where the results of algorithms [Nguyen and Chatterjee 2000; Potkonjak et al. 1996; Arfaee et al. 2009] were taken from Arfaee et al. [2009]. As can be observed from Table IX, HCMVM finds better solutions than all algorithms in terms of the number of operations, except that both H_{2MC} and HCMVM obtain the best solution on the DHT instance.

As the fifth experiment set, we used $k \times k$ DCTs, where k ranges from 4 to 20 in an increment of 2, and the constants were defined under bit-widths between 1 and 16. Table X presents the results of algorithms when constants are defined under CSD, where *avg*. and *CPU* present the average number of operations and CPU time in seconds, respectively.

Table X. Summary of Results of the Algorithms on $k \times k$ DCTs

Algorithm	[Hosangadi et al. 2005]		H_{2MC}		HCMVM	
DCTs	avg.	CPU	avg.	CPU	avg.	CPU
4 × 4	16.25	0.08	16.25	0.08	16.25	0.17
6 × 6	30.81	0.51	30.69	0.47	30.69	1.36
8 × 8	53.94	3.27	54.06	4.92	53.88	8.68
10 × 10	72.56	12.88	71.13	7.39	70.25	29.26
12 × 12	97.81	51.88	96.06	25.88	95.94	112.56
14 × 14	138.31	110.16	136.69	53.51	135.75	202.32
16 × 16	178.94	317.84	175.94	138.44	175.13	562.33
18 × 18	180.75	934.72	177.75	467.79	176.94	3368.99
20 × 20	227.44	2164.48	223.31	1100.24	221.75	5546.38
22 × 22	315.88	4768.48	310.56	2553.37	309.06	8927.03
24 × 24	323.00	8080.29	310.44	4336.18	309.75	28831.11

Table XI. Summary of High-Level Algorithms on 10 × 10 and 20 × 20 DCTs

bw	10 × 10 DCTs						20 × 20 DCTs					
	[Hosangadi et al. 2005]			HCMVM			[Hosangadi et al. 2005]			HCMVM		
	op	as	CPU	op	as	CPU	op	as	CPU	op	as	CPU
2	41	4	0.3	32	4	0.8	118	6	6.5	98	5	16.0
4	56	5	1.5	48	7	4.8	156	7	46.7	156	8	193.3
6	65	6	3.6	65	7	10.9	192	8	105.3	189	8	392.8
8	72	6	8.2	72	7	19.3	232	11	250.4	232	11	905.4
10	80	7	13.0	77	8	38.1	257	11	414.6	254	11	2135.8
12	92	8	17.7	89	8	39.1	300	13	624.7	295	13	3808.9
14	99	8	34.4	98	11	49.7	323	13	962.7	319	13	4814.8
16	112	9	46.5	110	10	111.8	376	15	1556.9	357	15	9002.1
Total	617	53	125.2	591	62	274.5	1954	84	3967.8	1900	84	21269.1

As can be observed from Table X, on small-size matrices, the algorithms obtain similar results. However, on larger-size instances, H_{2MC} obtains better solutions than the heuristic of Hosangadi et al. [2005]. The difference in the average number of operations between the results of these algorithms is 12.56 on 24 × 24 DCTs. On the other hand, HCMVM finds similar solutions to H_{2MC}, since the difference method cannot find valuable subexpressions that can be used for the implementation of linear transforms using fewer operations on DCTs. The reason is that the DCT matrices include a small number of different constants, which diminishes the advantage of using the difference method. Moreover, HCMVM takes more time to find a solution than the heuristics of Hosangadi et al. [2005] and H_{2MC}, due to the new differences obtained in each iteration.

6.3 Synthesis of Linear DSP Transforms

In this experiment, we used 10 × 10 and 20 × 20 DCTs, where the bit-width (bw) of the constants were defined from 2 bits to 16 bits with an increment of 2. Table XI presents the solutions of high-level algorithms when constants are defined under CSD.

Observe from Table XI that HCMVM finds better solutions in terms of the number of operations than the CSE heuristic [Hosangadi et al. 2005] on total, requiring 3.25 and 6.75 fewer operations on average on 10 × 10 and 20 × 20 DCTs, respectively. However, there are instances where both algorithms obtain a solution with the same number of

Table XII. Summary of Low-Level Results of the Algorithms on 10 × 10 DCTs

bw	Direct Realization			[Hosangadi et al. 2005]			HCMVM		
	area	delay	power	area	delay	power	area	delay	power
2	17.8	2496	0.6	13.4	2705	0.5	10.6	2646	0.4
4	32.1	3208	1.6	17.4	3330	0.9	14.9	3657	0.8
6	47.5	3257	2.3	20.7	3588	1.2	19.3	3726	1.2
8	64.7	3568	3.4	23.2	4077	1.5	21.5	4025	1.3
10	74.2	3573	3.8	26.4	4423	1.7	24.1	4294	1.6
12	82.4	3735	4.4	30.6	4592	2.0	27.2	4611	1.9
14	94.7	4046	5.2	32.5	5336	2.4	29.2	5420	2.2
16	89.8	4628	4.8	39.2	5107	3.1	33.7	5261	2.7
Total	503.2	28511	26.0	203.4	33158	13.3	180.5	33640	12.1

Table XIII. Summary of Low-Level Results of the Algorithms on 20 × 20 DCTs

bw	Direct Realization			[Hosangadi et al. 2005]			HCMVM		
	area	delay	power	area	delay	power	area	delay	power
2	68.4	2874	3.3	36.5	3202	2.0	29.6	3066	1.7
4	115.2	3712	7.5	49.3	4132	3.5	45.1	4015	3.3
6	160.8	3772	10.0	60.7	4176	4.6	53.8	4237	4.2
8	235.4	3772	14.6	76.3	5473	6.4	64.5	4971	5.6
10	271.2	3838	17.1	87.7	5435	7.7	73.6	5325	6.5
12	304.7	4029	19.8	101.9	5262	9.4	84.4	5704	7.6
14	346.8	4410	22.5	112.7	5837	10.8	94.3	5724	9.2
16	353.3	5074	22.8	122.5	5812	13.1	103.6	5846	11.0
Total	1855.8	31481	117.6	647.6	39329	57.5	548.7	38888	49.1

operations, that is, on 10 × 10 DCTs when *bw* is 6 and 8 and on 20 × 20 DCTs when *bw* is 4 and 8.

Tables XII and XIII present the low-level design results of 10 × 10 and 20 × 20 DCTs, respectively, which are synthesized using the Cadence Encounter® RTL Compiler with the Nangate 45nm Open Cell library.[10] In these tables, *area (mm^2)*, *delay (ps)*, and *power (mW)* indicate area, delay, and power dissipation, respectively. In this experiment, the bit-width of each element of the input vector was taken as 16. Also, in the *direct realization* of linear transforms, they were described as the additions/subtractions of constant multiplications in VHDL, and the solutions of the heuristic [Hosangadi et al. 2005] and HCMVM were translated into VHDL using only addition/subtraction and shift operations. The same design script was used for each VHDL code during the synthesis of DCT circuits.

Observe from Tables XII and XIII that the use of high-level algorithms targeting a shift-adds architecture leads to significant improvements in terms of area and power dissipation when their results are compared with those obtained using direct realizations of DCTs. Also, the solutions of HCMVM yield low-complexity DCT designs when compared to Hosangadi et al. [2005]. This is simply because, besides its better high-level results, some hardware optimizations are also considered in HCMVM. Its efficiency can be easily observed in the solutions of algorithms that have the same number of operations, that is, on 10 × 10 DCTs when *bw* is 6 and 8, and on 20 × 20 DCTs when *bw* is 4 and 8. We also note that the values of average and maximum area

[10]The gate library is available at http://www.nangate.com/.

improvements obtained by HCMVM over the CSE algorithm [Hosangadi et al. 2005] are 12.7% and 26.4% on 10×10 DCTs, respectively. These values are 18.0% and 23.4% on 20×20 DCTs, respectively.

7. CONCLUSIONS

In this article, we introduced optimization algorithms for the realization of linear transforms using the fewest number of addition and subtraction operations in a shift-adds architecture. We started by formalizing the CMVM problem as a 0-1 ILP problem and presented the complexity of the exact CSE algorithm in terms of 0-1 ILP problem parameters. Due to the NP-completeness of the problem, we introduced a CSE heuristic algorithm that finds the most common minimum conflicting 2-term subexpressions iteratively. Since the solutions of the CSE algorithms are restricted to a particular number representation, we also proposed a hybrid algorithm that combines the numerical difference method with the proposed CSE heuristic. It is observed from the experimental results on randomly generated constant matrices and linear DSP transforms that the proposed approximate algorithms find solutions close to the minimum and obtain better results than previously proposed efficient heuristics. The experimental results on the synthesis of DCTs also indicate that the solutions with the fewest number of operations obtained by the proposed hybrid algorithm lead to low-complexity and low-power DCT circuits.

REFERENCES

AKSOY, L., COSTA, E., FLORES, P., AND MONTEIRO, J. 2007. Optimization of area in digital FIR filters using gate-level metrics. In *Proceedings of the Design Automation Conference*. 420–423.

AKSOY, L., COSTA, E., FLORES, P., AND MONTEIRO, J. 2008. Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits 27*, 6, 1013–1026.

AKSOY, L., GUNES, E., AND FLORES, P. 2010. Search algorithms for the multiple constant multiplications problem: Exact and approximate. *Elsevier J. Microprocess. Microsyst. 34*, 151–162.

ALOUL, F., RAMANI, A., MARKOV, I., AND SAKALLAH, K. 2002. Generic ILP versus specialized 0-1 ILP: An update. In *Proceedings of International Conference on Computer-Aided Design*. 450–457.

ARFAEE, A., IRTURK, A., LAPTEV, N., FALLAH, F., AND KASTNER, R. 2009. Xquasher: A tool for efficient computation of multiple linear expressions. In *Proceedings of Design Automation Conference*. 254–257.

AVIZIENIS, A. 1961. Signed-digit number representation for fast parallel arithmetic. *IRE Trans. Electronic Comput. EC-10*, 389–400.

BARTH, P. 1995. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Tech. rep., Max-Planck-Institut fur Informatik.

BOULLIS, N. AND TISSERAND, A. 2005. Some optimizations of hardware multiplication by constant matrices. *IEEE Trans. Computers 54*, 10, 1271–1282.

CAPPELLO, P. AND STEIGLITZ, K. 1984. Some complexity issues in digital signal processing. *IEEE Trans. Acoustics, Speech, Signal Process. 32*, 5, 1037–1041.

COUDERT, O. 1996. On solving covering problems. In *Proceedings of Design Automation Conference*. 197–202.

DEMPSTER, A. AND MACLEOD, M. 1995. Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Trans. Circuits Syst. II 42*, 9, 569–577.

DEMPSTER, A. AND MURPHY, N. 2000. Efficient Interpolators and filter banks using multiplier blocks. *IEEE Trans. Signal Process. 48*, 1, 257–261.

DEMPSTER, A., GUSTAFSSON, O., AND COLEMAN, J. 2003. Towards an algorithm for matrix multiplier blocks. In *Proceedings of IEEE European Conference on Circuit Theory and Design*. IEEE, Los Alamitos, CA, 1–4.

EEN, N. AND SORENSSON, N. 2006. Translating pseudo-Boolean constraints into SAT. *J. Satisfiability, Boolean Model. Comput. 2*, 1–26.

ERCEGOVAC, M. AND LANG, T. 2003. *Digital Arithmetic*. Morgan Kaufmann.

FAUST, M. AND CHANG, C.-H. 2010. Minimal logic depth adder tree optimization for multiple constant multiplication. In *Proceedings of IEEE International Symposium on Circuits and Systems*. IEEE, Los Alamitos, CA, 457–460.

GARNER, H. 1965. Number systems and arithmetic. *Advances Comput. 6*, 131–194.

GUSTAFSSON, O. 2007. Lower bounds for constant multiplication problems. *IEEE Trans. Circuits Syst. II: Analog Digital Signal Process. 54*, 11, 974–978.

GUSTAFSSON, O., DEMPSTER, A., AND WANHAMMAR, L. 2002. Extended results for minimum-adder constant integer multipliers. In *Proceedings of IEEE International Symposium on Circuits and Systems*. 73–76.

GUSTAFSSON, O., OHLSSON, H., AND WANHAMMAR, L. 2004. Low-complexity constant coefficient matrix multiplication using a minimum spanning tree. In *Proceedings of Nordic Signal Processing Symposium*. 141–144.

HARTLEY, R. 1996. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Trans. Circuits Syst. II 43*, 10, 677–688.

HOSANGADI, A., FALLAH, F., AND KASTNER, R. 2005. Reducing hardware complexity of linear DSP systems by iteratively eliminating two-term common subexpressions. In *Proceedings of Asia and South Pacific Design Automation Conference*. 523–528.

HOU, H. 1987. A fast recursive algorithm for computing the discrete cosine transform. *IEEE Trans. Acoustics, Speech, Signal Process. 35*, 10, 1444–1461.

LARRABEE, T. 1992. Test pattern generation using Boolean satisfiability. *IEEE Trans. Computer-Aid. Des. Integrat. Circuits 11*, 1, 4–15.

LEFEVRE, V. 2001. Multiplication by an integer constant. Tech. rep., Institut National de Recherche en Informatique et en Automatique.

MUHAMMAD, K. AND ROY, K. 2002. A graph theoretic approach for synthesizing very low-complexity high-speed digital filters. *IEEE Trans. Computer-Aid. Des. Integrat. Circuits 21*, 2, 204–216.

NGUYEN, H. AND CHATTERJEE, A. 2000. Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis. *IEEE Trans.VLSI 8*, 4, 419–424.

PARK, I.-C. AND KANG, H.-J. 2001. Digital filter synthesis based on minimal signed digit representation. In *Proceedings of Design Automation Conference*. 468–473.

POTKONJAK, M., SRIVASTAVA, M., AND CHANDRAKASAN, A. 1996. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits 15*, 2, 151–165.

THONG, J. AND NICOLICI, N. 2009. Time-efficient single constant multiplication based on overlapping digit patterns. *IEEE Trans. VLSI Syst. 17*, 9, 1353–1357.

VORONENKO, Y. AND PÜSCHEL, M. 2007. Multiplier-less multiple constant multiplication. *ACM Trans. Algorithms 3*, 2.

WANG, Y. AND ROY, K. 2005. CSDC: A new complexity reduction technique for multiplier-less implementation of digital FIR filters. *IEEE Trans. Circuits Syst. 52*, 9, 1845–1853.

YURDAKUL, A. AND DÜNDAR, G. 1999. Multiplier-less realization of linear DSP transforms by using common two-term expressions. *J. VLSI Signal Process. 22*, 3, 163–172.

Received July 2010; revised May 2011; accepted July 2011