An Approximate Algorithm for the Multiple Constant Multiplications Problem

Levent Aksoy and Ece Olcay Gunes
Istanbul Technical University
Department of Electronics and Communication Engineering
Maslak, 34469, Istanbul, Turkey
{aksoyl, ece.gunes}@itu.edu.tr

ABSTRACT
Multiple constant multiplications (MCM) problem that is to obtain the minimum number of addition/subtraction operations required to implement the constant multiplications finds itself and its variants in many applications, such as finite impulse response (FIR) filters, linear signal transforms, and computer arithmetic. There have been a number of efficient algorithms proposed for the MCM problem. However, due to the NP-hardness of the problem, the proposed algorithms have been heuristics and cannot guarantee the minimum solution. In this paper, we introduce an approximate algorithm that can ensure the minimum solution on more instances than the previously proposed heuristics and can be extended to an exact algorithm using an exhaustive search. The approximate algorithm has been applied on a comprehensive set of instances including FIR filter and randomly generated hard instances, and compared with the previously proposed efficient heuristics. It is observed from the experimental results that the proposed approximate algorithm finds competitive and better results than the prominent heuristics.

Categories and Subject Descriptors
B.2.0 [Arithmetic and Logic Structures]: General.

General Terms
Algorithms, design.

Keywords
Multiple constant multiplications problem, graph-based technique, approximate algorithm, multiplierless filter design.

1. INTRODUCTION
In several computationally intensive applications, such as finite impulse response (FIR) filters as illustrated in Figure 1, and fast Fourier transforms, multiple constants are required to be multiplied by the same input. Since the design complexity of these applications is dominated by the constant multiplications, the shift-adds architecture, that is also the best option for maximum performance and minimum power consumption, is commonly used to implement constant multiplications [15]. When the same input is to be multiplied by a set of constant coefficients, significant reductions in hardware can also be obtained by sharing the partial products of the input among the set of multiplications. Thus, since shift operations are free in terms of hardware, the multiple constant multiplications (MCM) problem is defined as finding the fewest number of addition/subtraction operations to implement the constant multiplications. The MCM problem has been proven to be NP-hard in [3].

As an example of an MCM problem, suppose the multiple constant multiplications of 11 and 13 by the input \( x \), i.e., \( 11x \) and \( 13x \). The shift-adds implementations of constant multiplications are presented in Figure 2. Observe that while the multiplierless implementation without partial product sharing requires 4 operations, Figure 2(a), the sharing of partial product \( 9x \) in both multiplications reduces the number of operations to 3, Figure 2(b).

In the last decade, efficient algorithms have been proposed for the optimization of the number of operations in MCM. These methods can be categorized in two classes: common subexpression elimination (CSE) [13, 16] and graph-based
algorithms [7, 18]. The CSE algorithms, initially, define the constants under a particular number representation and then, obtain a solution by finding the most common non-zero digit patterns on the representations of the constants iteratively. For the example given in Figure 2, suppose the constants in multiplications 11x and 13x are defined in binary, i.e., $11x = (1011)_{bin}x$ and $13x = (1101)_{bin}x$ respectively. Observe that the sharing of partial product $9x$ is possible when the common partial product, i.e., $(1011)_{bin}x = x + 8x = 9x$, is identified in both multiplications. The exact CSE algorithms of [12, 9] model the MCM problem as a 0-1 integer linear programming problem and find the minimum number of operations solution of the MCM problem by maximizing the partial product sharing. It is argued in [8, 4] that being limited to a number representation does not yield the minimum number of operations solutions. These heuristics obtain much better solutions than the CSE heuristics by extending the possible implementations of constants based on a number representation. Also, the algorithm of [1] extends the exact CSE algorithm of [9] to handle the constants under the general number representation increasing the search space and finds more promising solutions than those of the exact CSE algorithm. On the other hand, the graph-based algorithms are not restricted to a particular number representation and synthesize the constants iteratively by building a graph.

Although the graph-based algorithms give better results than CSE algorithms as shown in [18], they are based on heuristics and provide no indication on how far from the minimum their solutions are. In a recent paper [10], the lower bound on the number of required operations to implement the MCM is investigated. However, the solution found by a heuristic is determined as minimum, if the number of operations in the found solution is equal to the lower bound. To the best of our knowledge, there is no exact graph-based algorithm designed for the MCM problem.

In this work, we introduce an approximate graph-based algorithm that guarantees the minimum solution on more instances than the prominent heuristics. The approximate algorithm is applied on FIR filter and randomly generated instances, and compared with efficient graph-based algorithms. It is observed from the experimental results that the number of instances that the approximate algorithm guarantees the minimum solution is greater than those of the previously proposed heuristics and it obtains competitive and better results than the efficient graph-based heuristics.

The rest of the paper is organized as follows: Section 2 gives the background concepts and Section 3 describes the approximate graph-based algorithm. Experimental results are presented in Section 4 and finally, the paper is concluded in Section 5.

2. BACKGROUND

In this section, initially, we present the main concepts and the problem definition, and then, we give an overview of the graph-based algorithms.

2.1 Definitions

In the MCM problem, the main operation, called the $A$-operation in [18], is an operation with two integer inputs, $u$ and $v$, and one integer output, $w$, that performs a single addition or a subtraction, and an arbitrary number of shifts.

![Figure 3: The representation of the $A$-operation in a graph.](image)

It is defined as follows:

$$w = A(u, v) = |(u \ll l_1) + (-1)^s(v \ll l_2)| \gg r = |2^{|i|}u + (-1)^s2^{|j|}v2^{-r}|$$

(1)

where $l_1, l_2 \geq 0$ are integers denoting left shifts, $r \geq 0$ is an integer indicating the right shift, and $s \in \{0, 1\}$ is the sign that denotes the addition/subtraction operation to be performed. The operation that implements a constant can be represented in a graph where the vertices are labeled with constants and the edges are labeled with the sign and shifts as illustrated in Figure 3. In the MCM problem, the complexity of an adder and a subtractor is assumed to be equal in hardware. It is also assumed that the sign of the constant can be adjusted at some part of the design and the shifting operation has no cost, since shifts can be implemented with only wires in hardware. Thus, in the MCM problem, only positive and odd constants are considered. Observe from (1) that in the implementation of an odd constant with the $A$-operation considering odd constants at the inputs, one of the left shifts, $l_1$ or $l_2$, is zero and $r$ is zero, or $l_1$ and $l_2$ are zero and $r$ is greater than zero. It is also necessary to constrain the left shifts, $l_1$ and $l_2$, to make the set of choices finite. As shown in [7], it is sufficient to limit the shifts by the maximum bit-width of the target constants and allowing larger shifts does not improve the solutions obtained with the former limits. However, in the proposed approximate algorithm and also, in the algorithm of [18], the number of shifts is allowed to be at most $bw + 1$. Thus, the MCM problem can be also defined as follows:

**Definition 1.** Given the target set $T = \{t_1, \ldots, t_n\} \subset \mathbb{N}$, including unrepeated positive and odd target constants, find the smallest ready set $R = \{r_0, r_1, \ldots, r_m\} \subset T$ such that $r_0 = 1$ and for all $r_k$ with $1 \leq k \leq m$, there exist $r_i, r_j$ with $0 \leq i, j < k$ and an operation $r_k = A(r_i, r_j)$.

Hence, the number of addition/subtraction operations required to implement the MCM is $|R| - 1$ [18].

2.2 Related Work

For the single constant multiplication problem, an exact algorithm that finds the minimum number of required operations for a constant up to 12 bit-width is introduced in [6] and is extended up to 19 bit-width in [11]. For the MCM problem, four algorithms, ‘add-only’, ‘add/subtract’, ‘add/d/shift’, and ‘add/subtract/shift’, are proposed in [2]. The latter algorithm, i.e., ‘add/subtract/shift’, is modified in [7], called BHM, by extending the possible implementations of a constant, considering only odd numbers, and processing constants in order of increasing single coefficient cost, that is evaluated by the algorithm of [6]. Also, a graph-based algorithm, called RAG-n, is introduced in [7]. RAG-n has two parts: optimal and heuristic. In the optimal part, each target constant that can be implemented with a single operation whose inputs are in the ready set are synthesized. If
there exist unimplemented element(s) left in the target set, the algorithm switches to the heuristic part where an intermediate constant is added to the target set. In the selection of the intermediate constant, RAG-n chooses an unimplemented target constant with the smallest single coefficient cost and synthesizes it with an intermediate constant that has the smallest value among the possible constants. Then, it removes the target constant to the ready set and inserts the intermediate constant to the target set. The graph-based heuristic of [18], called Hcub, includes the same optimal part of RAG-n, but uses a better heuristic that considers the impact of each possible intermediate constant on all target constants to be implemented and chooses the one that yields the best cumulative benefit. Also, Hcub is not restricted to the lookup table that includes single coefficient cost of constants as RAG-n, thus it is applicable to larger size constants. To the best of our knowledge, the algorithm of [18] finds solutions that are significantly better than the solutions found by any of the previously published algorithms.

3. THE APPROXIMATE ALGORITHM

As given in Section 2.1, the MCM problem can be defined as finding the minimum number of intermediate constants such that each constant, target and intermediate, can be synthesized with an operation as given in (1) where \( u \) and \( v \) are 1, target, or intermediate constants. Hence, the proposed approximate graph-based algorithm is designed to find the fewest number of intermediate constants such that each intermediate and target constant can be implemented using a single operation at the end of the algorithm rather than synthesizing the target constants once at a time by finding the “best” intermediate constant as done in previously proposed graph-based algorithms.

In the preprocessing phase of the algorithm, the target constants are made positive and odd, and added to the target set, \( T \), without repetition. The maximum bit-width of the target constants, \( bw \), is determined. In the main part of the approximate algorithm given in Algorithm 1, the ready set that includes the fewest number of elements is computed.

In the \texttt{ApproximateSearch}, initially, the ready set including only 1 is formed as given on the line 1 of the algorithm. Then, the target constants that can be implemented with the elements of the ready set using a single operation are found iteratively and removed to the ready set using the \texttt{Synthesize} function. If there exist unimplemented constant(s) in the target set, then in each iteration of the infinite loop, i.e., the line 6 of the algorithm, an intermediate constant is added to the ready set until there is no element left in the target set. The approximate algorithm considers the odd constants that are not included in the current ready and target sets and can be implemented with the elements of the current ready set as possible intermediate constants, as seen on the lines 7-10 of the algorithm. Note that the ready and target sets denoted by \( A \) and \( B \) represent the working ready and target sets respectively. Then, each possible intermediate constant is added to the working ready set and its implications on the current target set are found by the \texttt{Synthesize} function. If there exist unimplemented target constants in the working target set, the implementation cost of the unimplemented target constants is found in terms of the single coefficient cost evaluated in [11] and is assigned to the cost value of the intermediate constant, as given on line 17 of the algorithm. After the cost value of each intermediate constant is found, the one with the minimum cost is chosen to be added to the current ready set and the target constants that can be implemented with the elements of the current ready set are found. The infinite loop is interrupted whenever there is no element left in the working target set, thus the solution is obtained with the working ready set. However, note that by adding an intermediate constant to the ready set in each iteration, the previously added intermediate constants can be redundant due to the recently added constant. Hence, the \texttt{RemoveRedundant} function is applied on the obtained ready set to remove the redundant intermediate constants. After the ready set that consists of the fewest number of constants is obtained, each element in the ready set, except 1, are synthesized with a single operation whose inputs are the elements of the ready set.

\begin{algorithm}
\caption{The approximate algorithm.} \label{alg:approx}
\begin{algorithmic}
\STATE \textbf{ApproximateSearch}(T)
\STATE \hspace{1em} \textbf{if} \( T = \emptyset \) \textbf{then} \STATE \hspace{2em} return \( R \)
\STATE \hspace{1em} \textbf{else} \STATE \hspace{2em} \textbf{while} 1 \textbf{do} \STATE \hspace{3em} \textbf{for} \( j = 1 \) to \( 2^{bw+1} - 1 \) \textbf{step} 2 \textbf{do} \STATE \hspace{4em} \textbf{if} \( j \notin R \) \textbf{and} \( j \notin T \) \textbf{then} \STATE \hspace{5em} \textbf{(A, B) = Synthesize(R, \{j\})} \STATE \hspace{5em} \textbf{if} \( B = \emptyset \) \textbf{then} \STATE \hspace{6em} \( A \leftarrow A \cup \{j\} \)
\STATE \hspace{5em} \textbf{if} \( B \neq \emptyset \) \textbf{then} \STATE \hspace{6em} \textbf{(A, B) = Synthesize(A, T)} \STATE \hspace{5em} \textbf{if} \( B = \emptyset \) \textbf{then} \STATE \hspace{6em} \( A \leftarrow \text{RemoveRedundant}(A) \)
\STATE \hspace{5em} \textbf{return} \( A \)
\STATE \hspace{3em} \textbf{else} \STATE \hspace{4em} \textbf{cost} \leftarrow \text{EvaluateCost}(B) \STATE \hspace{4em} \textbf{if} \( \text{cost} \) \textbf{is minimum among all possible intermediate constants,} \( j \) \textbf{then} \STATE \hspace{5em} \textbf{R \leftarrow R \cup \{ ic \}} \STATE \hspace{5em} \textbf{(R, T) = Synthesize(R, T)} \STATE \hspace{2em} \textbf{end while} \STATE \hspace{1em} \textbf{end if} \STATE \hspace{1em} \textbf{end for} \STATE \hspace{1em} \textbf{end if} \STATE \hspace{1em} \textbf{end if} \STATE \hspace{1em} \textbf{end do} \STATE \hspace{1em} \textbf{end for} \STATE \hspace{1em} \textbf{Synthesize}(R, T)
\STATE \hspace{1em} \textbf{repeat} \STATE \hspace{2em} \textbf{isadded} \leftarrow 0 \STATE \hspace{2em} \textbf{for} \( k = 1 \) to \( |T| \) \textbf{do} \STATE \hspace{3em} \textbf{if} \( t_k \) \textbf{can be synthesized with the elements of} \( R \) \textbf{then} \STATE \hspace{4em} \textbf{isadded} \leftarrow 1 \STATE \hspace{3em} \textbf{R \leftarrow R \cup \{ t_k \}} \STATE \hspace{3em} \textbf{T \leftarrow T \setminus \{ t_k \}} \STATE \hspace{3em} \textbf{until} \textbf{isadded} \leftarrow 0 \STATE \hspace{2em} \textbf{return} \( (R, T) \)
\STATE \hspace{1em} \textbf{EvaluateCost}(T)
\STATE \hspace{1em} \textbf{cost} \leftarrow 0 \STATE \hspace{2em} \textbf{for} \( k = 1 \) to \( |R| \) \textbf{do} \STATE \hspace{3em} \textbf{if} \( r_k \) \textbf{is an intermediate constant} \textbf{then} \STATE \hspace{4em} \textbf{R \leftarrow R \setminus \{ r_k \}} \STATE \hspace{4em} \textbf{(R, T) = Synthesize(\{1\}, R)} \STATE \hspace{3em} \textbf{if} \( T \neq \emptyset \) \textbf{then} \STATE \hspace{4em} \textbf{R \leftarrow R \cup \{ r_k \}} \STATE \hspace{3em} \textbf{return} \( R \)
\end{algorithmic}
\end{algorithm}
We make some simple observations on the approximate algorithm. In these observations, \( |T| \) denotes the number of unrepeated positive and odd target constants to be implemented in the beginning of the algorithm, i.e., the lowest bound on the minimum number of operations solution.

**Lemma 1.** If the approximate algorithm finds a solution with \( |T| \) operations, then the found solution is minimum.

In this case, no intermediate constant is required to implement the target constants. Because the elements of the target set cannot be synthesized using less than \( |T| \) operations as shown in [7], if the approximate algorithm finds a solution including \( |T| \) operations, then the found solution is the minimum solution.

**Lemma 2.** If the approximate algorithm finds a solution with \( |T| + 1 \) operations, then the found solution is minimum.

In this case, only one intermediate constant is required to implement the target constants. Because the case described in Lemma 1 is checked on the lines 2-3 of the algorithm, if there exist unimplemented target constants, then the minimum solution requires at least one intermediate constant. So, if the solution found by the approximate algorithm includes \( |T| + 1 \) operations, then it is the minimum solution.

**Lemma 3.** If the approximate algorithm finds a solution with \( |T| + 2 \) operations, then the found solution is minimum.

In this case, two intermediate constants are required to implement the target constants. Because the case described in Lemma 1 is checked on the lines 2-3 of the algorithm and the case described in Lemma 2 is explored exhaustively on the line 7 of the algorithm, if there exist unimplemented target constants at the end of the first iteration, then the minimum solution requires at least two intermediate constants. So, if the solution found by the approximate algorithm includes \( |T| + 2 \) operations, then it is the minimum solution.

It is obvious that if the approximate algorithm finds a solution including more than \( |T| + 2 \) operations, then the approximate algorithm cannot guarantee the found solution is minimum, since all possible intermediate constant combinations including more than two constants are not explored exhaustively in the algorithm. However, observe that the bounds on the minimum number of operations solution of the approximate algorithm can be extended when the exhaustive search is applied on these cases, consequently, an exact algorithm can be designed for the MCM problem. We note that RAG-n and Hcub algorithms guarantee the minimum solution if the solutions found by these heuristics include the number of operations up to \( |T| + 1 \).

As a small example, suppose the target set including 287, 307, and 487. Figure 4 presents the results obtained by Hcub and the approximate algorithm. Observe from Figure 4(a) that since Hcub synthesizes target constants once at a time by including intermediate constants, the intermediate constants included for the implementation of target constants in the previous iterations of Hcub may not be shared in the implementation of the target constants in later iterations, although Hcub is particularly designed for this case. On the other hand, in each iteration of the approximate algorithm, an intermediate constant that can be implemented with the elements of the current ready set is added to the ready set. On this example, the intermediate constants 5 and 25 are added to the ready set in the first and second iteration respectively, Figure 4(b). The intermediate constant chosen in each iteration is the constant that implements more not-yet synthesized target constants with the elements of the current ready set using a single operation. Also, note that the target constants that can be implemented with the elements of the current ready set are removed from the target set to the ready set in each iteration. Hence, the approximate algorithm may obtain better solutions than Hcub. Observe from Lemma 3 that the approximate algorithm also ensures the minimum solution on this example.

4. EXPERIMENTAL RESULTS

In this section, we present the results of the approximate algorithm on FIR filter and randomly generated instances, and compare with those of the previously proposed graph-based heuristics of [7] and [18]. The graph-based heuristics were obtained from [17].

As the first experiment set, we used FIR filter instances where filter coefficients were computed with the remez algorithm in MATLAB. The specifications of filters are presented in Table 1 where: pass and stop are normalized frequencies that define the passband and stopband respectively; \#tap is the number of coefficients; and width is the bit-width of the coefficients. We note that the filter 11 was used as an example filter in [10, 5]. In this table, \( |T| \) denotes the number of unrepeated positive and odd filter coefficients, i.e., the lowest bound on the number of operations, and LBs indicates the lower bounds on the number of operations and the number of operations in series, generally known as adder-step, obtained by the formulas given in [10]. The results of the algorithms are also presented in Table 1 where adder denotes the number of operations and step indicates the number of adder-step.

As can be easily observed from Table 1, the approximate algorithm finds similar or better solutions than the graph-based heuristics, and according to the lemmas given in Section 3, it guarantees the minimum solution on 9 filter instances. The filter instances that the approximate algorithm cannot guarantee the minimum solutions are filters 1 and 11. However, we note that the solutions of the approximate algorithm on these filters are the minimum solutions ensured by exploiting all possible intermediate constant combinations including two constants exhaustively. Observe that Hcub finds similar results with the approximate algorithm, but it obtains worse solutions on filters 1, 2, and 6 than the approximate algorithm, and it can only determine its solution on filter 4 as the minimum solution. On the other hand, RAG-n and BHM obtain suboptimal results on all filter instances that are far from the solutions of the approximate algorithm. Also, observe that the lower bound on the number of required operations can only be used to determine the solution of the approximate algorithm on filter 6 as the minimum solution, although it is also proven to be minimum by the given lemmas. This is because the formula given in [10] computes a lower bound on the number of operations close to the lowest bound. Thus, this experiment clearly indicates that the approximate algorithm that guarantees the minimum solution on more instances is indispensable to ensure the minimum solution of the MCM problem.

As can be observed from Table 1, the approximate algorithm finds the fewest number of operations solution of a filter instance in a greater number of adder-step with respect to its lower bound, indicating, in general, the traditional tradeoff between area and delay. This is because the shar-
of instances that RAG-n and Hcub ensure the minimum solution is 701, i.e., 37% of the experiment set, and the number that the approximate algorithm guarantees the minimum solution than the approximate algorithm is 173 on overall 688, while the number of instances that Hcub obtains better solutions than Hcub is 1890 instances. Also, we note that the number of instances that the approximate algorithm finds better solutions than Hcub is 701, i.e., 37% of the experiment set, and the number of instances that the proposed approximate algorithm finds better solutions than Hcub is 173 on overall 688, while the number of instances that RAG-n and Hcub ensure the minimum solution than the approximate algorithm is 173 on overall 688, while the number of instances that the approximate algorithm guarantees the minimum solution than the previously proposed heuristics. We have applied the approximate algorithm on a comprehensive set of instances and it is shown by the experimental results that the proposed approximate algorithm finds competitive and better solutions than the previously proposed graph-based heuristics. Also, it is shown that the approximate algorithm guarantees the minimum solution on more instances than the previously proposed graph-based heuristic of [18] that finds significantly better solutions than any other previously published graph-based heuristics.

5. CONCLUSIONS

In this paper, we introduce an approximate graph-based algorithm that finds the fewest number of intermediate constants such that the target and intermediate constants can be synthesized using a single operation at the end of the algorithm, rather than synthesizing the target constants once at a time by including intermediate constants. The design of the approximate algorithm in this scheme allows the algorithm to guarantee the minimum solution on more instances than the previously proposed heuristics. We have applied the approximate algorithm on a comprehensive set of instances and it is shown by the experimental results that the proposed approximate algorithm finds competitive and better solutions than the previously proposed graph-based heuristics. Also, it is shown that the approximate algorithm guarantees the minimum solution on more instances than the previously proposed graph-based heuristic of [18] that finds significantly better solutions than any other previously published graph-based heuristics.

6. REFERENCES

Figure 5: Comparison of algorithms on randomly generated hard instances.

(a) Randomly generated instances in 10 bits
(b) Randomly generated instances in 12 bits
(c) Randomly generated instances in 14 bits
(d) Randomly generated instances in 16 bits


