Time-Multiplexed Multiple-Constant Multiplication

Peter Tummeltshammer, *Student Member, IEEE*, James C. Hoe, *Member, IEEE*,
and Markus Püschel, *Senior Member, IEEE*

***Abstract*— This paper studies area-efficient arithmetic circuits to multiply a fixed-point input value selectively by one of several preset fixed-point constants. We present an algorithm that generates a class of solutions to this time-multiplexed multiple-constant multiplication problem by "fusing" single-constant multiplication circuits for the required constants. Our evaluation compares our solution against a baseline implementation style that employs a full multiplier and a lookup table for the constants. The evaluation shows that we gain a significant area advantage, at the price of increased latency, for problem sizes (in terms of the number of constants) up to a threshold dependent on the bit-widths of the input and the constants. Our evaluation further shows that our solution is better suited for standard-cell ASICs than prior works on Reconfigurable Multiplier Blocks (ReMBs).**

***Index Terms*— Multiplierless, reconfigurable multiplier block, addition chain**

I. Introduction

This paper is concerned with arithmetic circuits for the *time-multiplexed multiple-constant multiplication* with the interface shown in Fig. 1 and is an extension of our preliminary work in [1]. The input x is a fixed-point value of a specified bit-width. The output of the circuit is $c_i x$ where c_i is one of N given fixed-point constants $\{c_1, c_2, \ldots, c_N\}$ selected according to the $\lceil \log_2 N \rceil$-bit control input i. A straightforward implementation would comprise a lookup table of the N constants and a full multiplier (depicted later in Fig. 5(a)). Intuitively, the full generality of a full multiplier is unnecessary for multiplying by a predetermined set of constants. This paper presents a class of circuits that take advantage of the redundancy and structure in the constituent constants to reduce hardware cost in terms of the number and the size of the adders and multiplexers in the circuit. This problem (sometimes referred to as Reconfigurable Multiplier Blocks, or ReMBs) has been studied by prior work [2], [3], [4], [5], [6], [7] that develops solutions targeted to Field Programmable Gate Array (FPGA) implementations. Our approach uses a different architecture than these approaches and is optimized for ASIC implementations. We address the difference between our method and the prior work in Section II-B.

Our solution is based on "fusing" single-constant multiplication circuits of the required constants. Area-efficient arithmetic circuits to multiply a fixed-point input by one fixed-point constant have been studied extensively. The best approach for minimizing the number of required additions (and subtractions) is reported in [8] (more details are in Section II-A). Other constructions of single-constant multiplication circuits are possible. For example, for a given a set of constants, one can consider single-constant circuits that share intermediate values. Algorithms that produce these circuits are used for the related problem of parallel multiple-constant multiplication, e.g., [9], [10], [11].

Fig. 1. The time-multiplexed multiple-constant multiplier block considered in this paper. The input x is multiplied by one out of N given fixed point constants $c_1, \ldots, c_N$, based on the control input i.

Given an input set of single-constant multiplication circuits (in a graph representation), the algorithm in this paper derives an area-efficient circuit for the time-multiplexed multiple-constant multiplication. Specifically, given the N separate single-constant circuits for $\{c_1, c_2, \ldots, c_N\}$ as input, the N single-constant circuits are integrated iteratively into a "fused" circuit consisting of adders and multiplexers. The properly synthesized controls for the multiplexers enable the circuit to behave as a single-constant circuit for each of the N constants. The algorithm takes into consideration the commonalities in the topology of the N initial single-constant circuits and uses a heuristic search to produce a final fused circuit with the smallest bit-width adders and with the minimum number of steering multiplexers. The fused circuit contains only as many adders as the largest of the single-constant circuits. The number of multiplexers, however, depends on the constants and how the single-constant circuits are fused, i.e., which adders are multiplexed.

We implemented the fusion algorithm as a generator that takes as input the N constants and the bit-width of x, and generates the Verilog netlist of an area-optimized fused multiplication circuit for these constants. (The generator is available online at [12].) We first evaluate our algorithm on inputs of single-constant circuits generated using [8]. These circuits have the minimal number of adders. The evaluation includes a detailed comparison of the generated circuits against a default implementation based on multipliers and lookup tables. For small values of N, our solution yields considerably smaller synthesized circuit area in a commercial 0.18μm standard cell technology. However, for sufficiently large values of N, the fused multiplier circuits eventually grow to be too expensive. Namely, the area cost grows approximately linearly with N, in

Peter Tummeltshammer is with the Department of Computer Science, University of Technology, Vienna, Austria. Email: petertu@ecs.tuwien.ac.at. James C. Hoe and Markus Püschel are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh. E-mail: {jhoe,pueschel}@ece.cmu.edu .

This work was supported by a fellowship of the University of Technology, Vienna, for short-term scientifc research abroad, by NSF through awards 0325687 and 0310941, and by DARPA through the Department of Interior Grant NBCH1050009.

contrast to the baseline approach where only the constant table expands with N. For example, assuming a 16-bit input value and random 16-bit constants, our approach results in a smaller synthesized circuit area for up to 15 constants. The price for these gains is an increased latency compared to the baseline. We also evaluate our algorithm on inputs of single-constant circuits prepared according to [11]. These circuits share more intermediate values, but have in general more adders. As a consequence, we get only marginal improvements and only under very limited conditions. Nevertheless, in practice, both starting inputs for a given set of constants could be evaluated in conjunction since the runtime of our algorithm is typically only a few seconds. Finally, we compare our multiplier blocks for time-multiplexed multiple-constant multiplication against previous work on ReMB designs. For standard cell synthesis, we can demonstrate an area advantage that becomes more significant with increasing number of constants.

Paper outline. Following this brief introduction, Section II discusses additional background and related work on the problem of multiplying a value by one or several constants. Section III presents our algorithm for generating a time-multiplexed multiple-constant multiplier circuit by iteratively fusing single-constant circuits. We analyze the runtime of our algorithm and derive bounds on the quality of the generated circuits. Section IV evaluates our approach against the baseline approach and compares to previous work. Finally, we offer conclusions in Section V.

II. Multiplication by Constants

In this section we review the problem of multiplying an input by one or several fixed-point constants using only additions and shifts and put our contribution into the context of previous work. Further, we review the directed acyclic graph (DAG) representation of single- and multiple-constant multiplication, which provides a suitable way to formulate the problem and its solution.

Without loss of generality, we assume that the multiplicative constants are fixed-point positive integer numbers. The constant bit-width is denoted by w such that for a constant c,

$$c = b_{w-1}b_{w-2}\cdots b_1 b_0 = \sum_{i=0}^{w-1} b_i 2^i, \quad b_i \in \{0,1\}. \qquad (1)$$

A. Multiplication By One Constant

From (1) we see that the product cx of an input x and a w-bit constant c is given by $\sum_{i=0}^{w-1} b_i 2^i x$. This summation can be directly mapped into a series of shifts (scalings by powers of 2) and additions. In particular, if the binary representation of c has k non-zero digits, the multiplication cost in terms of the number of additions is $k-1$. In this paper, we assume the cost of shifts is negligible since they can be implemented as wired connections in hardware. For brevity, we use in the remainder of the paper the term "addition" for both additions and subtractions since these operations are virtually identical in complexity and map to similar hardware structures.

CSD representation. Both software and hardware compilers generally use signed digit (SD) recoding [13, chap. 6] to reduce the number of additions associated with multiplying by a constant. An SD constant is $b_{w-1}b_{w-2}\cdots b_0 = \sum_{i=0}^{w-1} b_i 2^i$ where $b_i \in \{\bar{1}, 0, 1\}$ and $\bar{1}$ stands for -1. The most salient aspect of SD recoding is in replacing the occurrences of k consecutive 1's by $10\cdots0\bar{1}$, which yields a saving of $k-2$ additions. The canonical signed digit (CSD) recoding requires in addition that no two consecutive bits are nonzero; CSD recoding results in a 33% cost saving on average [13, chap. 6].

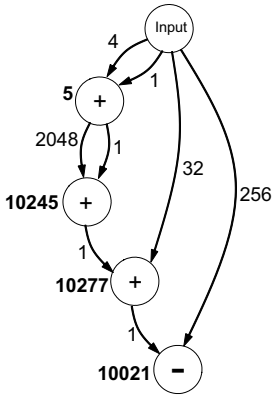


Fig. 2. The DAG corresponding to multiplying by 10021.

DAG based methods. The CSD method is not optimal in general. For example, multiplying an input x by $c = 10021_{10} = 10011100100101_2$ can be done in the following steps:

$$\begin{aligned} s_1 &= x + (x \ll 2) \\ s_2 &= s_1 + (s_1 \ll 11) \\ s_3 &= s_2 + (x \ll 5) \\ cx &= s_3 - (x \ll 8) \end{aligned} \qquad (2)$$

In this example, 4 additions are required to compute cx, compared to 5 additions required by the CSD method.

The computation in (2) is best represented as the directed acyclic graph (DAG) in Fig. 2. The nodes of the graph represent additions and the edges represent the dataflow between additions. We assume that each node has an in-degree of 2, i.e., each addition has two operands. Each edge is labelled by a positive 2-power integer $k = 2^s$, which represents the scaling (by shifting by s) applied to the operand on that edge. Each node is labelled by the intermediate constant f. These intermediate constants are referred to as the *fundamentals* in the DAG, following the terminology in [14]. If x is the DAG input and f is the fundamental of a node, then the intermediate output produced at this addition node is fx.

The problem of finding an optimal DAG for a constant is known to be NP-hard [15] and has been studied in the literature [14], [8]. It is related to, but in theory and solution fundamentally different from the addition chain problem, which considers DAGs without shifts [16, pp. 465–485] and subtractions. Recently, [8] has developed an algorithm that finds optimal DAGs for constants up to a maximal bit-width of 19, and shows that 5 additions are sufficient in all cases. In this paper, we use a re-implementation of this method. Fig. 3 shows the histogram of the number of adders for all odd 20-bit constants.

By propagating shifts, each DAG can be transformed into

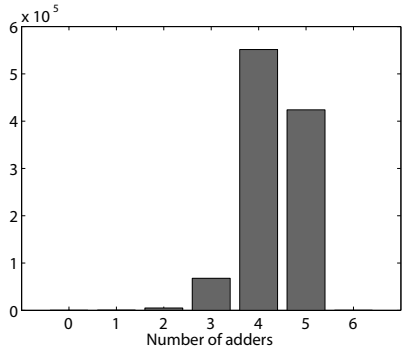


Fig. 3. Histogram for the number of adders in SCM DAGs for all odd 20-bit constants. Only one constant (699829) needs 6 adders.

an equivalent *normalized* DAG with equal cost that satisfies the following properties (see [14, theorem 2]):

- for each addition node, one of the operands is not shifted and, as a consequence,
- all fundamentals are odd.

For example, the DAG in Fig. 2 is normalized. In the remainder of this paper, we will consider only normalized DAGs.

Finally, for our fusion algorithm we assume that all the shifts in the single-constant DAGs are *left* shifts. This restriction may yield suboptimal DAGs, but only in very few cases. For example, we evaluated (using the tool in [11]) that among the 524288 odd 20-bit constants only 341 (or 0.065%) can save exactly one adder by using right shifts.

B. Multiplying by Multiple Constants

In this section, we consider the problem of multiplying an input x by a given set $\{c_1, \ldots, c_N\}$ of N constants again using only additions and shifts. There are two fundamentally different scenarios:

- *Parallel multiplication* is performed by a (parallel) multiplier block that simultaneously outputs the N values $c_1x, \ldots, c_Nx$. This problem is commonly referred to as *multiple-constant multiplication* or *MCM*.
- *Time-Multiplexed multiplication* is performed by a multiplier block that outputs c_ix as controlled by an input that specifies i (Fig. 1).

Both scenarios have been studied in the literature. The second is the subject of this paper. We briefly discuss both.

Parallel multiplication. The problem of multiplying an input by several constants in parallel occurs for example in finite impulse response (FIR) filters and thus was the subject of a large number of papers, e.g., [15], [9], [17], [18], [10]. One of the best available methods is [11], also available as online tool at [12]. Most of the above methods also use a DAG-based approach. The basic idea is to achieve savings by generating for the given constants DAGs that "overlap" for additional savings, which is a form of common sub-expression elimination. The goal is to minimize the total number of adders shared by all constants. The resulting multiplier block is shown as a schematic in Fig. 4. The leaf-shaped structures are the single-constant DAGs for $c_1, \ldots, c_N$. The overlapping regions signify common subexpressions or fundamentals. One of the overlapping regions is highlighted.

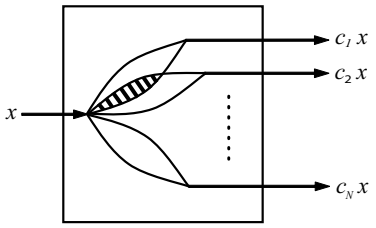


Fig. 4. Multiplier block for the parallel multiplication by N given constants.

Time-multiplexed multiplication. In this paper, we are interested in developing an area-efficient combinational logic block that can multiply a fixed-point input value by one of the N preset constants according to a $\lceil \log_2 N \rceil$-bit control input (Fig. 1). The most straightforward implementation of this logic block is shown in Fig. 5(a) where the preset constants are stored in a lookup table. At a given time step, the control input selects which constant is supplied to one input of a full multiplier.

Again the question arises whether it is possible to exploit redundancy or inherent structure of the problem using a DAG based approach. An immediate idea is to adapt the solution for the parallel multiplication (Fig. 4) to the sequential problem as depicted in Fig. 5(b), which uses a multiplexer to select one of the output products according to the control input. Whether this yields any savings compared to the generic solution depends on the number N of multiplicative constants and the degrees of overlapping between the individual DAGs. However, a coarse analysis already shows that the adapted solution in Fig. 5(b) is in general suboptimal for the sequential problem. The reason is that each unique fundamental would be instantiated as one adder. Since only one product is visible through the output multiplexer at any moment, the results of some adders are unused.

To further improve, one can fuse the fundamentals from different DAGs, equal or not, to share the same adders by time-multiplexing, thus exploiting the topological similarities between the DAGs to a much larger degree. This is the basic idea behind our solution depicted in Fig. 5(c), which inserts multiplexers into the DAG for time-multiplexed sharing of adders and thus reduced area requirements. Our solution employs only as many adders as required by the largest of the initial DAGs being fused.

Related work. Time-multiplexed multiple-constant multiplication — sometimes referred to as Reconfigurable Multiplier Block (ReMB) — has been studied extensively in [2], [3], [4], [5], [6], [7], [19] in the context of FPGA implementations. The architecture of these prior solutions is motivated by the fact that in the construction of an adder in an FPGA, a 2-to-1 multiplexer can be inserted in front of one input at no additional cost. A stand-alone 2-to-1 multiplexer would incur a cost comparable to an adder of the same width.

Thus, the basic building block in these solutions performs the operation $SUM = A + (select\ ?\ B : C)$. For example, the circuit in shown later in Fig. 15(a) (repro-

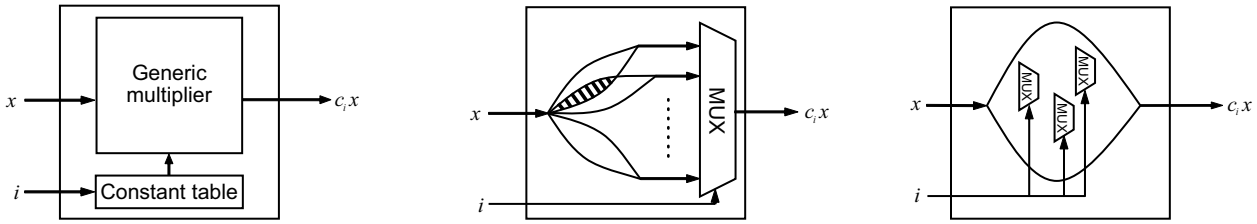


Fig. 5. Three implementations (as schematics) of the multiplier block in Fig. 1. From left to right: The standard solution using a generic multiplier; an adaptation of the parallel multiplier block in Fig. 4; proposed method with shared adders.

duced from Fig. 6 in [5]) is an ReMB for the constants $\{1, 2, 4, 8, 15, 26, 50, 162, 256\}$. However, the ReMB approach is not optimal in general because the FPGA-motivated constraints lead to solutions that use more than the minimal possible number of adders. For example, the aforementioned ReMB comprise 7 adders, although the optimal DAGs for the individual constants need no more than 2 adders each. On non-FPGA technologies, there is opportunity to reduce the number of the adders (by time-multiplexed reuse) at the cost of additional (but relatively cheaper) multiplexers. For example, for the same set of constants, the approach in this paper will produce a circuit with only 2 adders. As a part of our evaluation in Section IV, we compare the synthesized standard-cell circuit area resulting from our approach based on time-multiplexed reuse to ReMB examples from [3], [5], [7]. One should also point out that the straightforward implementation in Fig. 5(a) is, in most scenarios, the most economical solution on FPGAs given the availability of embedded multipliers and hard memory macros [20].

III. DAG Fusion

This section describes the proposed fusion algorithm. The input to the algorithm is a set of N normalized DAGs representing single-constant multiplications for $c_1, \ldots, c_N$; the output is a composite, or fused, DAG that consists exclusively of additions, shifts, and multiplexers and implements the time-multiplexed multiplication by $c_1, \ldots, c_N$ as depicted in Fig. 5(c). One crucial property of our fusion algorithm is that the number of additions in the generated composite DAG is equal to the largest number of additions required by any of the input DAGs, and therefore does not grow with N.

A simple example. Before starting the technical explanation of the DAG fusion algorithm, we show a small example. In Fig. 6, the two DAGs in (a) and (b) are optimal for multiplying by 45 and 19, respectively. Each of these DAGs requires 2 additions, and thus the composite DAG, produced by our algorithm and shown in (c), also requires $\max\{2, 2\} = 2$ additions. With both multiplexers set to select their left inputs, the active datapaths in this composite DAG correspond to DAG (a), and with both multiplexers set to their right input it corresponds to DAG (b). Note that in the fusion process, we replaced 2 additions by 2 multiplexers. Since a multiplexer requires about half as much combinational area as an addition (in ASIC), this saves area compared to the competing multiplierless solution Fig. 5(b), which, in this case, does not allow for DAG overlapping since the DAGs in Fig. 6(a) and (b) have no common fundamentals.

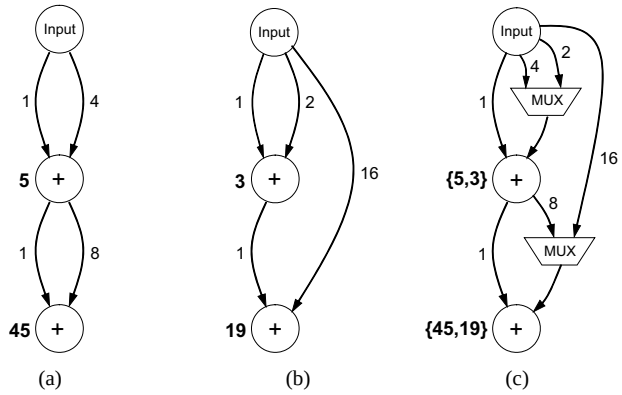


Fig. 6. (a) Optimal DAGs for 45; (b) optimal DAG for 19; and (c) the fused DAG produced by our algorithm.

In the remainder of this section we provide the details of the fusion algorithm. We start with an algorithm for the baseline case of fusing $N = 2$ DAGs. This algorithm is then used iteratively, as a subroutine primitive, in the algorithm for the fusion of N DAGs.

A. Fusing Two DAGs

We start with 2 DAGs denoted as DAG_L and DAG_R, for the multiplication with constants c_L and c_R, respectively. We assume that DAG_L and DAG_R have n and m nodes (or additions) and denote the node sets as follows:

$$\begin{aligned} \mathrm{Nodes}_L &= \{\mathrm{Node}_{L,0}, \mathrm{Node}_{L,1}, \ldots, \mathrm{Node}_{L,n-1}\}, \\ \mathrm{Nodes}_R &= \{\mathrm{Node}_{R,0}, \mathrm{Node}_{R,1}, \ldots, \mathrm{Node}_{R,m-1}\}. \end{aligned}$$

Without loss of generality, we assume $n \geq m$.

Intuitively, the fusion algorithm tries to find and exploit similarities in the topology of the two DAGs. These similar regions are fused and allow the additions in DAG_L and DAG_R to time-multiplex the same adder instantiations with little hardware overhead. In regions that are not similar, adders can still be time-multiplexed by the additions in DAG_L and DAG_R, but multiplexers must be inserted to connect the correct input sources to the shared adders or to correct for different shifts of the addition's operands.

The algorithm: Overview. A high level description of the fusion procedure for 2 DAGs, called FusePairDags, is shown in Algorithm 1. The algorithm enumerates all admissible

Algorithm 1 Fusing two DAGs. Input: Two DAGs $\mathrm{DAG}_L, \mathrm{DAG}_R$ for multiplying with two constants c_L, c_R; the DAGs have n and m nodes (additions) respectively; $n \geq m$. Output: Fused DAG with small area estimate for the time-multiplexed multiplication with c_L and c_R.

FusePairDAGs($\mathrm{DAG}_L,\ \mathrm{DAG}_R$)

```
best_area = ∞
best_dag = nil
for all admissible assignments φ of DAG_L to DAG_R do
    dag = FusePair(DAG_L, DAG_R, φ)
    area = EstimateArea(dag)
    if area < best_area then
        best_area = area
        best_dag = dag
    end if
end for
return best_dag
```

assignments of Nodes_R to Nodes_L. Admissible means that the assignment respects the ordering of the nodes in both DAGs. For each such assignment, the function FusePair merges the edges of DAG_L and DAG_R in the best possible way w.r.t. the area estimation function EstimateArea, which is explained later in Section III-B. Finally, the composite DAG with the lowest estimated area among all enumerated assignments is returned.

We provide additional detail on the node assignment and on the edge merging procedure. Then we slightly extend FusePairDags to the case where DAG_L is already composite. This extension will serve as the subroutine in the iterative step when merging $N > 2$ DAGs.

Node assignment. To fuse DAG_L and DAG_R, one must assign each node $\mathrm{Node}_{R,i}$ in DAG_R to a unique node $\mathrm{Node}_{L,j}$ in DAG_L, which means $\mathrm{Node}_{R,i}$ and $\mathrm{Node}_{L,j}$ will share the same addition in the composite DAG. Each assignment is an injective mapping $\phi:\ \mathrm{Nodes}_R \rightarrow \mathrm{Nodes}_L$, i.e., no two nodes in DAG_R are mapped to the same node in DAG_L. Further, to allow fusion, ϕ has to respect the partial orderings imposed by the dependencies in the two DAGs. In the simplest and most common case, both DAG_L and DAG_R are totally-ordered (e.g., Fig. 2), which means there are $\binom{n}{m}$ possible assignments ϕ, which FusePairDags will enumerate. With respect to node assignment, the most expensive DAGs have minimum depth relative to the number of nodes, since this produces the least ordering constraints. For a single-constant DAG with m nodes, the minimal possible depth is $\lceil \log_2(m+1) \rceil$; Fig. 7 shows an example of such a DAG. In this worst case scenario, the number of admissible ϕs is $n!/(\lceil \log_2(m+1) \rceil !(n-m)!)$. For the constant bit-widths ≤ 20 considered in this paper, $n, m \leq 6$ as shown in Fig. 3; thus, the number of admissible ϕs cannot exceed 720 and does not impose a computational problem.

Merging Edges. Assume an assignment ϕ of nodes has been fixed. The function FusePair then creates a composite DAG starting from the input node. Consider the fusion of two addition nodes $\mathrm{Node}_{R,i}$ and $\mathrm{Node}_{L,j}$. Remember that each

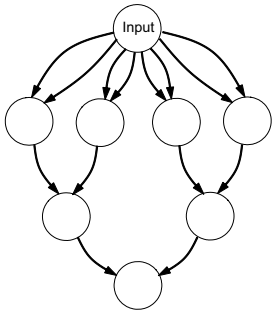


Fig. 7. Structure of a worst case minimum-depth DAG.

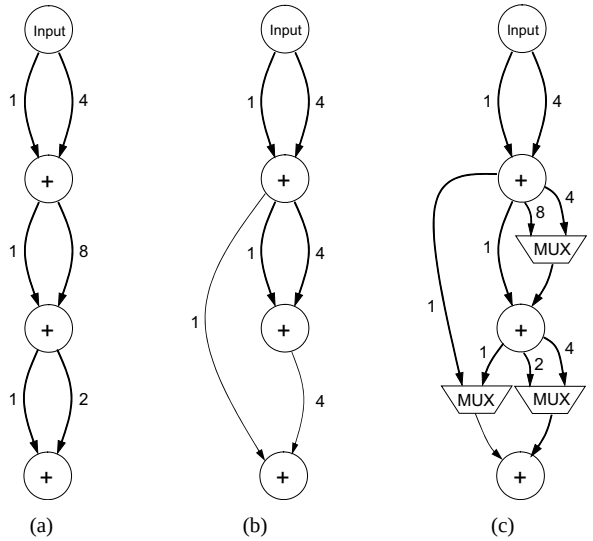


Fig. 8. Effect of merging edges: (a) and (b) are merged into (c). The fusion of the upper adders produces no multiplexer, the middle adders produce one, while the lower adders produce two multiplexers.

node has two operands (incoming edges) and that one of them is not shifted. One of three cases applies at each node as illustrated in Fig. 8:

- The two incoming edges of the corresponding Node_L and Node_R are shifted by the same values *and* belong to the same operand (a fused common predecessor node). Then the composite DAG does not need any multiplexers for this node. For example, this case occurs for the upper adders in Fig. 8.
- Exactly one of the two incoming edges of the corresponding Node_L and Node_R is shifted by the same value and belongs to the same operand. In this case one multiplexer is needed in the composite DAG for this node to accommodate for the remaining dissimilar edges. For example, this case occurs for the middle adders in Fig. 8 as well as for both adders in Fig. 6.
- In all other cases, two multiplexers are needed for fusing the corresponding Node_L and Node_R to accommodate for different input shifts and/or different operands. For example, this case occurs for the lower adders in Fig. 8.

Note that FusePair also tries to flip the incoming edges of a node (since addition is commutative) to improve the result. Thus, at most $2m$ fusions are tried for each call of FusePair.

Subtractions. A DAG may contain subtractions. If two

subtraction nodes are fused, the result is again a subtraction node. If an addition node and a subtraction node are fused, the result is a combined addition/subtraction node, supported by most macro libraries. Also, if a node is not a pure addition node, commutativity is lost, and FusePair does not flip edges to try to improve area.

Extension to support composite DAG$_L$. To use FusePairDags for the iterative fusion of N DAGs, we have to generalize it to the case of a DAG_L that is already composite, i.e., that may contain multiplexers before each node.

The node assignment in this case is the same as in the previous case. The edge merging in FusePair is also very similar. In fact, the chances of efficient merging are now increased as each node in DAG_L now has several incoming multiplexed edges. In case a node with a v-to-1 multiplexer is fused, and a new multiplexer must be added, it becomes a $(v+1)$-to-1 multiplexer.

Introducing larger multiplexers increases the effort for finding an optimal edge assignment, because when adding a new edge to a multiplexer, the multiplexer has to be checked if the edge already exists. The cost of this pass is $O(v)$ for a v-to-1 multiplexer.

B. Fusing Multiple DAGs

In this section we explain the algorithm for fusing N constant DAGs in the general case $N \geq 2$. The algorithm is essentially an iterative fusion of 2 DAGs (Algorithm 1) combined with a search over different orderings of fusing these DAGs. We analyze the impact of reordering the DAGs and explain the area estimation function (already used in Algorithm 1) to select the best DAG among the generated alternatives.

The algorithm. Pseudocode for fusing N DAGs is shown in Algorithm 2 called FuseNDags. The input is an array of N DAGs, representing the multiplications by the N constants $c_1, \ldots, c_N$, and an integer parameter Num_Iterations. The output is a composite DAG that multiplies by c_i, $1 \leq i \leq N$, according to a $\lceil \log_2 N \rceil$-bit control input.

The algorithm loops for Num_Iterations to evaluate fusing the input DAG array in different randomly chosen orderings. In each iteration, the randomly permuted input array is fused using the function FusePairDags in Algorithm 1 repeatedly. The lowest cost DAG among all different orderings is returned. The cost is again estimated using the cost function EstimateArea.

Effect of ordering. First, we provide an example that shows why the order of fusion makes a difference. Fig. 9(a), (b), (c) shows three DAG nodes that are to be fused. Fusing (a) and (b) first yields (d); then fusing (d) and (c) yields (f). Fusing (a) and (c) first yields (e); then fusing (e) and (b) yields (g), which has the same functionality as (f).

We observe that (g) needs one 3-to-1 multiplexer and one 2-to-1 multiplexer, whereas (f) needs only two 2-to-1 multiplexers and thus less logic. This difference arises in FuseNDags because FusePairDags makes a local decision about where multiplexers are inserted, and these decisions can impact the options available to subsequent calls to FusePairDags for the remaining DAGs. In this example, when fusing (a) and (c), two equal cost outcomes are possible; the connections of 2^4y and 2^0y could be switched in (e). Unfortunately, the option taken in (e) forces a 3-to-1 multiplexer when attempting to further fuse (b). Taking the other option (not shown) would have led to the same optimal result as in (f).

To overcome the problem of possibly choosing a clearly suboptimal solution due to the wrong fusion order, FuseNDags enumerates Num_Iterations many orderings. The maximum number of orderings possible is $N!$, which makes an exhaus-

Algorithm 2 Fusing N DAGs. Input: A list $\mathrm{DAG}[N]$ of N DAGs for constants $c_1, \ldots, c_N$. Output: Fused DAG with low area estimate for the time-multiplexed multiplication with $c_1, \ldots, c_N$.

FuseNDAGs($\mathrm{DAG}[N]$, Num_Iterations)

```
1:  best_cost = ∞
2:  best_dag = nil
3:  for i = 1, ..., Num_Iterations do
4:     randomly permute DAG[N]
5:     dag = DAG[1]
6:     for j = 2, ..., N do
7:        dag = FusePairDAGs(dag, DAG[j])
8:     end for
9:     cost = EstimateArea(dag)
10:    if cost < best_cost then
11:       best_cost = cost
12:       best_dag = dag
13:    end if
14: end for
15: return best_dag
```

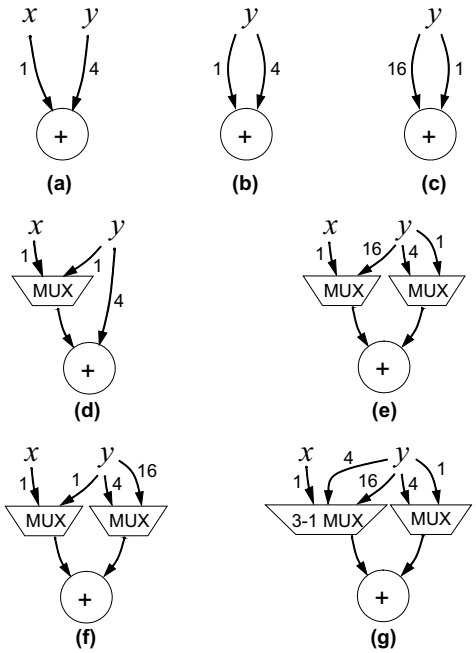


Fig. 9. Different orders to fuse three DAGs fragments. (a), (b) and (c) are three DAG fragments to be fused. (d) and (f) show the progression of first fusing (a) and (b) and then (c); (e) and (g) show the progression of first fusing (a) and (c) and then (b).

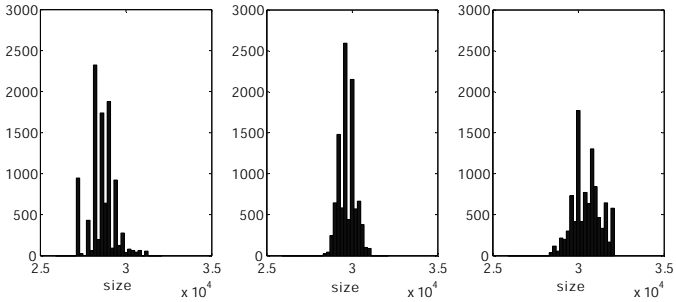


Fig. 10. Representative histograms of estimated areas. Each histogram shows the area distribution (estimated) of 10,000 out of 16! possible different ordering to fuse 16 randomly selected DAGs with five additions.

tive search eventually impractical for large N. However, for example for $N = 8$, FuseNDags only needs on the order of seconds on a current workstation to iterate through all possible fusions of 8 random 16-bit constant DAGs.

To analyze the expected gain obtained by enumerating different orderings, we conducted an experiment on ten different random sets of sixteen 16-bit constants, each requiring 5 additions (the maximum possible). In each case we fused the DAGs using 10,000 random orderings and evaluated the results using the area estimation function EstimateArea explained below. Fig. 10 gives the histograms of three representative trials among the ten. The difference between the best-case ordering and worst-case ordering is about 10–15% in each case. Thus, the expected gain of considering a number of different orderings is about 5–8%.

Finally, we note that we explored a "greedy scheme", which determined an order of fusion, by fusing those DAGs first that have the largest overlap when fused with FusePairDags. However, the method did not improve over selecting a random order, so we omit a more detailed explanation.

Area estimation. Both algorithms FusePairDags and FuseNDags use the area estimation function EstimateArea. This function makes one pass through a given composite DAG and recursively computes the bit-widths needed for each of the occurring multiplexers, adders, subtractors, and adder/subtractors. Knowing the bit-width, say k, for any of these building blocks, the area can then be estimated in square micron as $a \cdot k$, where a is a constant depending on the ASIC technology and library used for mapping. In our case, we mapped to a 0.18μm ASIC technology, using Synopsys Design Compiler v. 2002.05-SP2 and a commercial 0.18μm standard cell library, optimizing for area. The constants a were estimated, respectively, as follows:

- v-to-1 Multiplexer: $a_{mux} = 14v$;
- Adder: $a_{add} = 67$;
- Subtractor: $a_{sub} = 75$;
- Adder/Subtractor: $a_{addsub} = 98$.

The bit-widths bw for each block in the composite DAG are computed recursively, starting from the input node, which incurs cost 0. The total DAG cost A is then obtained by multiplying all bit-widths with the respective constants a_* above and summing them: (3).

$$A = \sum_{i=1}^{n_{mux}} a_{mux} \cdot bw_{mux_i} + \sum_{i=1}^{n_{add}} a_{add} \cdot bw_{add_i} + \sum_{i=1}^{n_{sub}} a_{sub} \cdot bw_{sub_i} + \sum_{i=1}^{n_{addsub}} a_{addsub} \cdot bw_{addsub_i}. \quad (3)$$

We assume we fused N DAGs and the fused DAG has an input bit-width of n. In the recursive cost computation on the composite DAG, we distinguish three cases for the current block:

- *v-to-1 Multiplexer.* The multiplexer has v inputs, each corresponding to the input value times the predecessor fundamental f_i and scaled by a shift s_i, for $1 \le i \le v$. The ith input value needs $n + \log_2(f_i) + s_i$ bits to represent. The multiplexer bitwidth needs to handle the largest input value among the v inputs. We compute the maximum input bit-width using the auxiliary terms
$$\begin{aligned} f_{\text{mux}} &= \max_{1\le i\le m}(f_i 2^{s_i}), \\ s_{\text{mux}} &= \min_{1\le i\le m}(s_i), \end{aligned} \quad (4)$$
and get as bit-width for the multiplexer
$$bw_{\text{mux}} = \log_2(f_{\text{mux}}) + n - s_{\text{mux}}.$$
s_{mux} is subtracted because for a non-zero s_{mux}, the least-significant bits to the multiplexer is hardwired to 0 for all inputs. The numbers f_{mux} and s_{mux} are also used for the computations below.
- *Adder.* Following [14],
$$bw_{\text{add}} = \max(\log_2(f_{\text{lmux}}) + s_{\text{lmux}}, \log_2(f_{\text{rmux}}) + s_{\text{rmux}}) + n - \max(s_{\text{lmux}}, s_{\text{rmux}}), \quad (5)$$
where lmux and rmux denote the left and right predecessor multiplexer, respectively, and the values f, s are obtained from (4).
- *Subtractor or Adder/Subtractor.* As in (5), but without the term $\max(s_{\text{lmux}}, s_{\text{rmux}})$ because one cannot hardwire those least significant bits as zero in a subtraction as in an addition.

We carefully compared the area estimate computed by EstimateArea with the post-synthesis area. The average error was below 10% in all cases.

C. Analysis

In this section we analyze the number of adders and multiplexers of the fused DAGs generated by our proposed Algorithm 2 and its asymptotic runtime.

Quality of generated fused DAG. We consider N distinct single-constant DAGs to be fused and denote with $A_{\max}$ the largest number of adders in any of these DAGs. Further, we denote with $A(N)$ and $M(N)$ the number of adders and the number of multiplexers in the composite DAG. By multiplexer we refer to a 2-to-1 multiplexer; an n-to-1 multiplexer is assumed to be built from $n - 1$ many 2-to-1 multiplexers. We have

$$A(N) = A_{\max}, \quad \text{and} \quad (6)$$
$$\lceil \log_2(N) \rceil \le M(N) \le (2A_{\max} - 1)(N - 1). \quad (7)$$

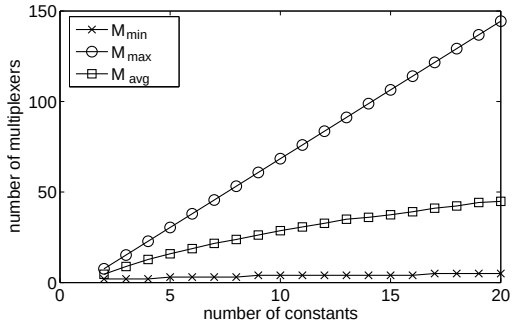


Fig. 11. Bounds and average number of multiplexers in DAG fusion for maximum constant bit-width of 16.

Equality (6) holds since our fusion algorithms does not add any adders but only multiplexers.

The lower bound in (7) can be derived from the fact that every multiplexer added to a DAG can at most double the number of possible outcomes. The upper bound in (7) assumes that in the worst case every fused adder introduces two multiplexers. The question may arise why $(2A_{max}-1)$ is used instead of $2A_{max}$. The reason is that the first fundamental in every single-constant multiplication DAG has an unshifted and a shifted edge to the input node. The unshifted edge can be used for all other fused DAGs, thus requiring one multiplexer less for each DAG.

The average number of multiplexers depends on the structure of the input DAGs and thus cannot be easily analyzed.

Fig. 11 shows the upper bound, the lower bound and the average number of multiplexers. The latter was determined empirically by fusing randomly drawn constants with a maximum bit-width of 16 and counting the multiplexers in the final design. Each point in M_{avg} is averaged over 100 experiments.

Runtime. The computational cost for fusing two DAGs DAG_L and DAG_R using Algorithm 1 can be divided into the cost for assigning nodes, the cost for merging edges, and the cost for evaluating the cost function.

Assigning nodes means finding the best node mapping ϕ for two DAGs with n and m additions respectively, and requires $O(n!/(\lceil \log_2(m+1) \rceil !(n-m)!))$ operations proportional to the number of these mappings. For $m = n$ this number is highest. In this paper we only consider $m, n \leq 6$; thus, the maximum number of possible ϕs is 720.

In Section III-A the edge merging cost has been shown to be $O(2m) = O(m)$ for m nodes on two single-constant DAGs. For a DAG DAG_L that has been obtain by the previous fusion of N single-constant DAGs, this cost increases due to the increased number of multiplexer options. In the worst case, there are $O(mN)$ such options since the largest multiplexer possible in DAG_L is now an N-to-1 multiplexer.

Evaluating the cost function on a DAG with n additions requires $O(3n) = O(n)$ operations, since the cost function makes exactly one pass through the fused DAGs, every addition can have a maximum of two ν-to-1 multiplexers, and the evaluation does not depend on ν.

This leads to the following overall runtime for fusing two DAGs. The first DAG has n additions and was obtained by the previous fusion of N DAGs. The second DAG has m additions and $n \geq m$.

$$\text{Runtime(FusePairDAGs)} = O\left(\frac{n!(mN+n)}{\lceil \log_2(m+1) \rceil !(n-m)!}\right) \quad (8)$$

Algorithm 2 repeats FusePairDAGs Num_Iterations times. Thus, fusing N DAGs using Algorithm 2 requires

$$\text{Runtime(FuseNDAGs)} = O(\text{Num\_Iterations} \cdot N \cdot \text{Runtime(FusePairDAGs)}), \quad (9)$$

with Runtime(FusePairDAGs) in (8). Note, that in Algorithm 2 the cost function does not produce any additional computation, since the DAGs' estimated costs are already known from FusePairDAGs.

TABLE I
RUNTIME OF ALGORITHM 2 IN SECONDS FOR VARIOUS NUMBERS OF CONSTANTS AND CONSTANT BITWIDTHS.

	number of constants N				
bitwidth w	4	8	12	16	20
4	0.005	0.005	0.008	0.01	0.014
8	0.006	0.015	0.02	0.02	0.04
12	0.006	0.076	0.33	0.6	1.54
16	0.008	0.573	2.06	6.8	46.0
20	0.313	1.057	2.97	18.9	102.9

To give an idea of the actual runtimes of the implemented algorithm, we ran benchmarks for all combinations of N constants of bitwidth w with $N \in \{4, 8, 12, 16, 20\}$ and $w \in \{4, 8, 12, 16, 20\}$ (see Table I). We used Num_Iterations = 100. Different values change the runtime roughly proportionally. The runtimes are in seconds and averages over 20 uniformly drawn sets of constants. For a fixed set of constants, the runtime may differ from this average since it depends on the complexity (number of adders and structure) of the single-constant DAGs. The results show that within the space of parameters considered, the runtime is negligible in a real-world design flow.

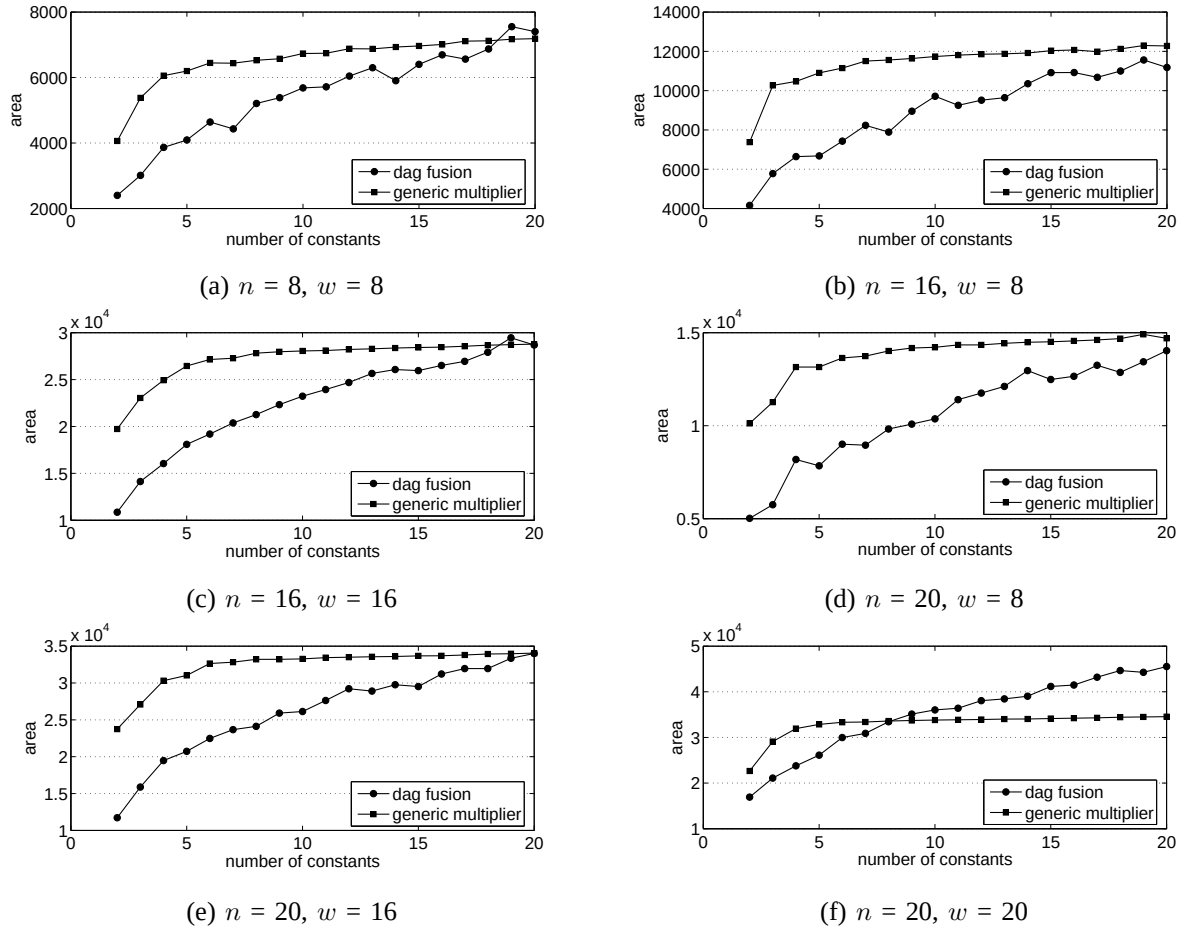
IV. EXPERIMENTAL RESULTS

This section presents an experimental evaluation of Algorithm 2. First, we apply Algorithm 2 to optimal single-constant DAGs and compare the generated solutions to the baseline implementation comprising a full multiplier and a constant table (Fig. 5(a)). Next, we apply Algorithm 2 to single-constant DAGs derived from parallel multiplier blocks [11] of the form in Fig. 4. These single-constant DAGs have maximized common fundamentals. Finally, we compare the performance of our generated multiplier blocks against Reconfigurable Multiplier Blocks found in the literature [3], [5], [7].

A. Fusing Optimal Single-Constant DAGs

The problem of time-multiplexed multiple-constant multiplication is parameterized by

- $n =$ bit-width of the input to the multiplication block;



(a) n = 8, w = 8

(b) n = 16, w = 8

(c) n = 16, w = 16

(d) n = 20, w = 8

(e) n = 20, w = 16

(f) n = 20, w = 20

Fig. 12. Average synthesized area of fused DAGs and generic multiplier solutions for varying input bit-width n and constant bit-width w.

- $w =$ maximum bit-width of the constants considered;
- $N =$ number of constants to be fused.

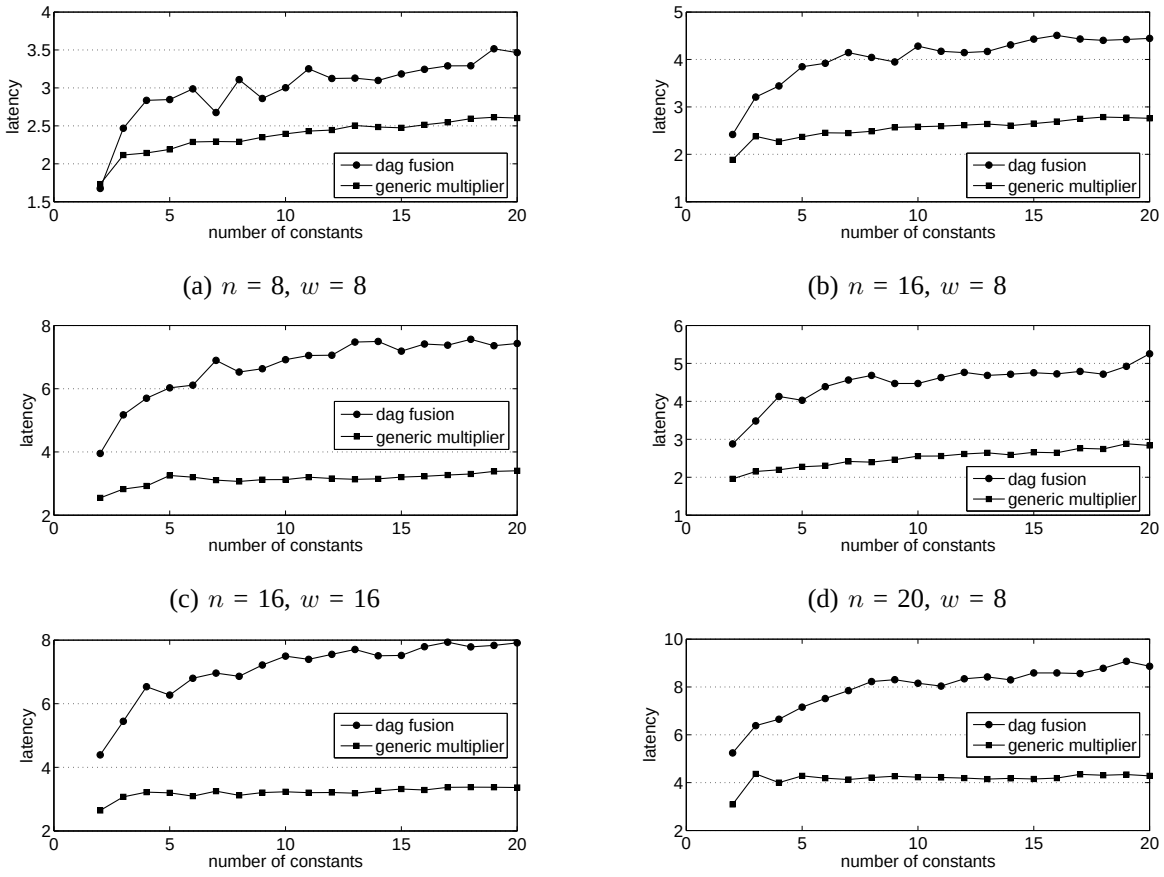
In this first set of evaluations, we compare the output of Algorithm 2 against the table-based baseline design (Fig. 5(a)) for all combinations of $n \in \{8, 12, 16, 20\}$, $w \in \{8, 12, 16, 20\}$ with $w \leq n$, and $N \in \{2, \ldots, 20\}$. For each combination of n, w, and N, the comparison is based on the average result over 10 sets of random constants uniformly drawn and with Num_Iterations $= 100$. All evaluated designs are described using structural Verilog and synthesized (optimizing for area) using the Synopsys Design Compiler (v. 2002.05-SP2) for a commercial 0.18μm standard cell library.

Area. Fig. 12 reports the resulting average synthesized areas (in square microns) as a function of N for different combinations of n and w. In the figures, the baseline designs based on generic multipliers follow a characteristic trend where the area cost increases sharply for the first 2 to 3 constants and then grows slowly afterwards. This is because for a small number of multiplicative constants, Synopsys can flatten the design for logic minimization. For a larger number of constants, the multiplier and table structure are left intact, and thus only the table structure grows as $O(N)$.

On the other hand, for the range of n, w and N considered, the area cost of the multiplier blocks produced by Algorithm 2 from optimal single-constant DAGs grows approximately linearly with N. In all figures, the generated multiplier blocks offer an advantage over the baseline for small values of N. However, as the number of constants increases, the linear growth in size of these blocks eventually exceeds the cost of the baseline design. Furthermore, the crossover value for N decreases for increasing w due to the larger, less fusible initial DAGs. Table II shows all values of N for those crossover points.

It is interesting to note that the DAG fusing approach is likely to be ineffective on modern field-programmable gate array (FPGA) architectures. Firstly, modern FPGAs offer dense hardwired multiplier macro blocks; this makes a multiplierless solution less attractive. Second, the resource cost of an adder is comparable to a multiplexer on an FPGA so that Algorithm 2 yields little gain if a new multiplexer is required to remove an adder.

Delay. Fig. 13 reports the resulting average synthesized critical path delay (in nanoseconds) corresponding to the data points in Fig. 12. The critical paths for the generated multiplier blocks are approximately two times greater than the corresponding baseline designs in all cases. From a performance standpoint, one should always utilize the baseline design especially when optimized multiplier macros cells are available.



(a) n = 8, w = 8 (b) n = 16, w = 8

(c) n = 16, w = 16 (d) n = 20, w = 8

(e) n = 20, w = 16 (f) n = 20, w = 20

Fig. 13. Average synthesized critical path delay of fused DAGs and generic multiplier solutions for input bit-width n and constant bit-width w.

TABLE II
CROSSOVER POINTS FOR INPUT BIT-WIDTH n AND CONSTANT BIT-WIDTH w

	constant bit-width w			
input bit-width n	8	12	16	20
8	19	–	–	–
12	>20	18	–	–
16	>20	19	19	–
20	>20	>20	20	9

B. Fusing Single-Constant DAGs Derived from Parallel Multiplication DAGs

Fig. 4 sketches a DAG for the parallel multiple-constant multiplication (MCM). For brevity, we call such a DAG MCM-DAG. The algorithm in [11] generates an MCM-DAG by maximizing the number of common fundamentals between the DAGs of different constants to minimize the total number of adders in the MCM-DAG. From this MCM-DAG, one can extract individual single-constant DAGs by pruning the unused portion of the MCM-DAG with respect to a given constant. The resulting set of single-constant DAGs then can be fused by Algorithm 2 to get a multiplier block for the multiplexed multiple-constant multiplication.

Because the MCM-DAG is globally optimized over the set of constants, a given derived single-constant DAG has in general more additions than an optimal single-constant DAG from [8]. Consequently, the resulting fused DAG (using Algorithm 2) will in general require more adders as well. On the other side, the increased number of common fundamentals may lead to a reduction of the number of multiplexers in the generated multiplier block. We evaluate this tradeoff in the following.

We compare the results of fusing MCM-derived single-constant DAGs versus fusing optimal single-constant DAGs (Section IV-A) over the problem space spanned by $n = w$, $w \in \{4, \ldots, 20\}$, $N \in \{2, \ldots, 20\}$. For each combination of w and N, the comparison is based on the average area over 100 sets of randomly selected constants using Num_Iterations=100. We determine the area cost using the cost function described in Section III-B. Fig. 14 displays the result. The horizontal axes correspond to w and N, respectively; the vertical axis is the area advantage of the MCM-derived DAGs in %. The region where the MCM-derived DAGs have an advantage is shaded in dark gray. The figure shows that gains can only achieved for a small number of constants N or for short constant bitwidths w. In other words, in these cases, the saving of using fewer multiplexers in fusion overcomes the cost of the extra adders of the MCM-DAGs.

In practice, both methods, i.e., both sets of DAGs, can be

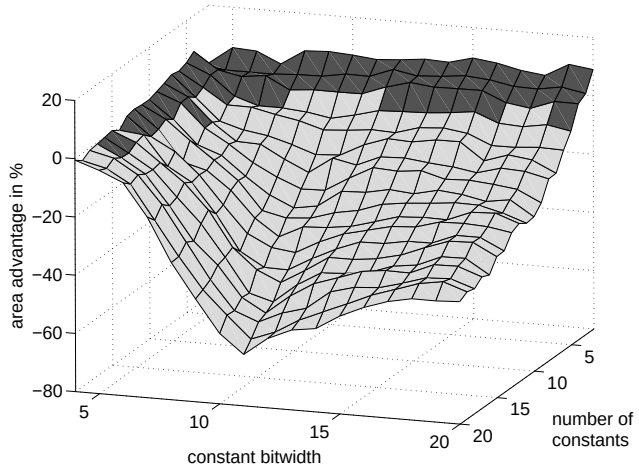


Fig. 14. Area advantage of fusing MCM-derived DAGs over fusing optimal DAGs.

TABLE III
THE SYNTHESIZED AREA OF OUR SOLUTIONS (FIG. 16) VERSUS THE CORRESPONDING REMBS (FIG. 15).

	multiplier block label			
	(a)	(b)	(c)	(d)
ReMB (Fig. 15)	16437	9203	7888	5361
DAG Fusion (Fig. 16)	6852	8859	6844	5222

tried due to the fast runtime (see Table I) of Algorithm 2.

C. Comparison Against Reconfigurable Multiplier Blocks

In this section we compare our results to the previous work on Reconfigurable Multiplier Blocks (ReMBs) for multiplexed multiple-constant multiplication (discussed earlier in Section II-B). Fig. 15 reproduces four ReMB examples from Fig. 6 of [5], Figs. 7 and 8 of [3] and Fig. 6 of [7]. The ReMB approach is geared towards FPGAs. Namely, the ReMB solutions use in general more adders than the minimal number possible in order to map efficiently to the special topology of FPGAs. (In particular, a 2-to-1 multiplexer comes free with each adder in FPGA technologies.)

In the example in Fig. 15(a), the ReMB solution uses 5 more adders than our corresponding solution with only 2 adders (in Fig. 16(a)) based on fusing optimal single-constant DAGs. The remaining three smaller ReMB examples each require 1 more adder than the fused-DAG solutions. When mapped onto an FPGA, our solutions are likely to consume more logic resources because we use more multiplexers, which are nearly as expensive as adders on FPGAs. Furthermore, our multiplexers usage do no adhere to any special topology and therefore cannot be absorbed for free into the construction of an adder in an FPGA.

However, when synthesizing for standard cell technologies, the FPGA-specific advantage leveraged by the ReMB algorithm no longer holds. Table III summarizes the synthesized area of ReMBs (Fig. 15) versus our solutions (Fig. 16) for the

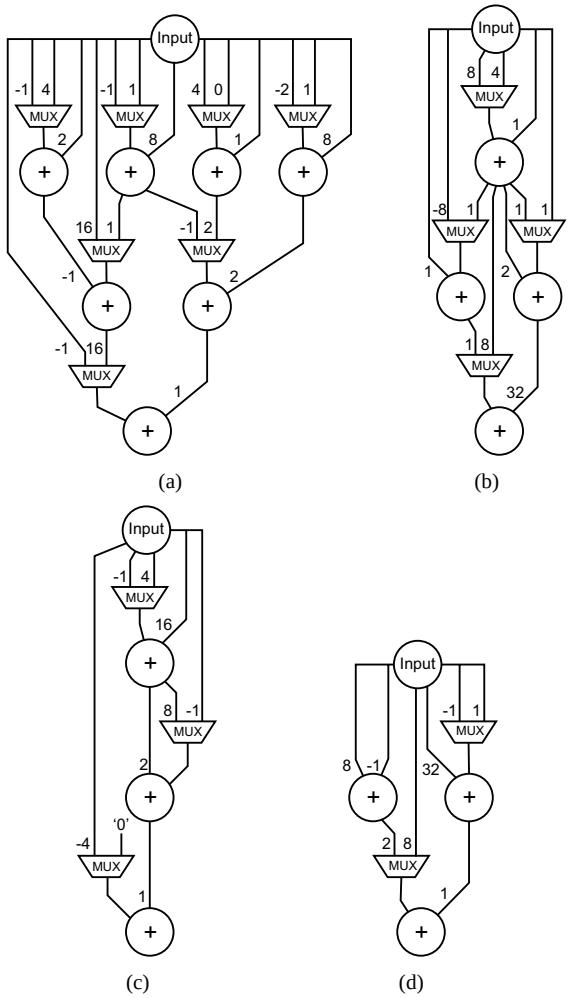


Fig. 15. Example ReMB solutions from (a) Fig. 6 of [5], (b) and (c) Figs. 7 and 8 of [3], and (d) Fig. 6 of [7]. In (c), the bottom mux has one constant-zero input.

example sets of constants. We see that in synthesized standard-cell technology, our tradeoff between adder and multiplexer costs leads to implementations with less area overall. We project from this evaluation that for larger constants, i.e., larger number of adders, this advantage becomes more pronounced.

V. CONCLUSIONS

We have presented an algorithm to construct area-efficient arithmetic circuits for time-multiplexed multiple-constant multiplication, that is, for the multiplication of a fixed-point input by one of several preset fixed-point constants according to a control input. The starting point to our algorithm is the DAG-based representations of circuits for multiplying by a single constant. The algorithm reduces the hardware resources by a process we have termed "fusion," i.e., by time-multiplexing the additions required for different constants to reuse the same adders in the final fused circuit. This keeps the number of adders in the generated blocks constant and only increases the multiplexers used.

Our algorithm is presented in detail including an analysis of its time complexity and the quality of its solutions. We showed that the generated multiplier blocks offer an area advantage

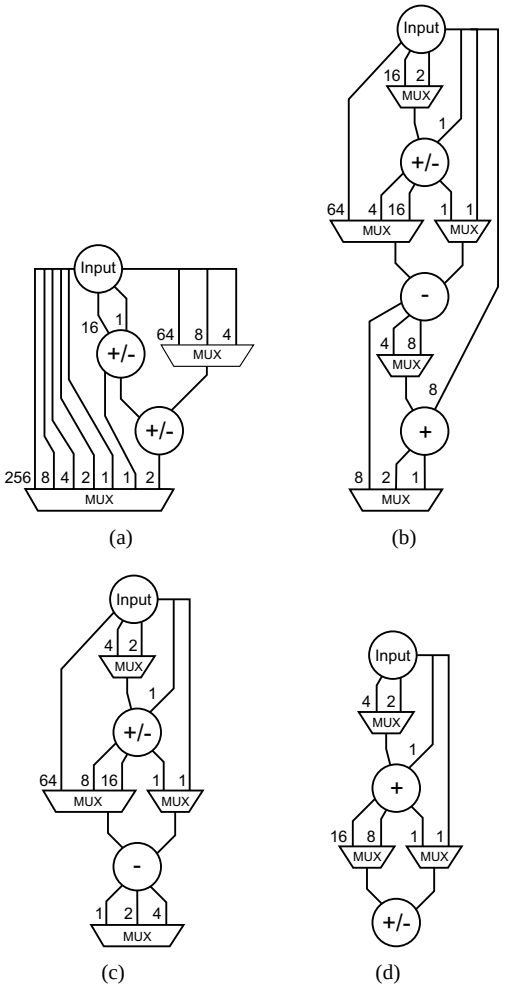


Fig. 16. Our solutions, produced by Algorithm 2, for the same sets of constants as the ReMB examples in Fig. 15.

over a conventional approach based on a constant table and a full multiplier across a range of relevant bitwidths and number of constants for the penalty of an increased latency. Further, the multiplier blocks produced from optimal (minimum number of adders) single-constant DAGs are better than those generated from single-constant DAGs optimized for common subexpressions. Finally, we demonstrated the advantage of our approach when targeting synthesized standard cell technology against the previous work on Reconfigurable Multiplier Blocks.

It is conceivable to further improve on our results using various heuristics. For example, the optimal single-constant DAG for a constant is not unique. Considering several or all possibilities in tandem with our algorithm may yield further gains. Another possibility would be to consider single-constant DAGs extracted from MCM-DAGs that have been optimized for minimum latency, i.e., that have fewer adders on the critical path. A different interesting direction is an extension of our method to produce multiplexed multiplier blocks for several outputs as in [6], [7] for FPGAs. These blocks can be used for small matrix-vector multiplications and thus, for example, as building blocks in linear transforms.

REFERENCES

[1] P. Tummeltshammer, J. Hoe, and M. Püschel, "Multiple constant multiplication by time-multiplexed mapping of addition chains," *Design Automation Conference*, vol. 41, pp. 826–829, Jun. 2004.

[2] R. Turner, T. Courtney, and R. Woods, "Implementation of fixed DSP functions using the reduced coefficient multiplier," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, May 2001, pp. 881–884.

[3] S. Demirsoy, A. Dempster, and I. Kale, "Design guidelines for reconfigurable multiplier blocks," in *International Symposium on Circuits and Systems*, vol. 4, May 2003, pp. 293–296.

[4] R. Turner and R. Woods, "Highly efficient, limited range multipliers for LUT-based FPGA architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 10, pp. 1113–1118, Oct 2004.

[5] S. Demirsoy, I. Kale, and A. G. Dempster, "Efficient implementation of digital filters using novel reconfigurable multiplier blocks," *Signals, Systems and Computers*, vol. 1, pp. 461–464, 2004.

[6] S. Demirsoy, I. Kale, and A. Dempster, "Synthesis of reconfigurable multiplier blocks: Part I - fundamentals," in *IEEE International Symposium on Circuits and Systems*, vol. 1, May 2005, pp. 536–539.

[7] ——, "Synthesis of reconfigurable multiplier blocks: Part - II algorithm," in *IEEE International Symposium on Circuits and Systems*, vol. 1, May 2005, pp. 540–543.

[8] O. Gustafsson, A. Dempster, and L. Wanhammar, "Extended results for minimum-adder constant integer multipliers," in *IEEE International Symposium on Circuits and Systems*, vol. 1. IEEE, May 2002, pp. I–73–I–76.

[9] A. G. Dempster and M. D. Macleod, "Use of minimum-adder multiplier blocks in FIR digital filters," *IEEE Transactions on Circuits and Systems II*, vol. 42, no. 9, pp. 569–577, 1995.

[10] P. Flores, J. Monteiro, and E. Coista, "An exact algorithm for the maximum sharing of partial terms in multiple constant multiplications," in *Proc. International Conference on Computer-Aided Design*, 2005, pp. 13–16.

[11] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms*, accepted for publication.

[12] "Spiral website," online: www.spiral.net.

[13] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A. K. Peters, 2001.

[14] A. G. Dempster and M. D. MacLeod, "Constant integer multiplication using minimum adders," *IEE Proceedings on Circuits, Devices and Systems*, vol. 141, no. 5, pp. 407–413, 1994.

[15] D. Bull and D. Horrocks, "Primitive operator digital filters," *IEE Proceedings G*, vol. 138, no. 3, pp. 401–412, 1991.

[16] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2, ch. 4.

[17] O. Gustafsson and L. Wanhammar, "A novel approach to multiple constant multiplication using minimum spanning trees," in *Proc. 45th Midwest Symposium on Circuits and Systems*, vol. 3, 2002, pp. 652–655.

[18] O. Gustafsson, H. Ohlsson, and L. Wanhammar, "Improved multiple constant multiplication using a minimum spanning tree," in *IEEE Conference on Signals, Systems and Computers*, vol. 1, 2004, pp. 63–66.

[19] N. Sidahao, G. A. Constantinides, and P. Y. Cheung, "Multiple restricted multiplication," *14th Int. Conf. Field Programmable Logic and Application (FPL'04)*, pp. 374–383, Aug./Sept. 2004.

[20] *Xilinx Virtex-II Pro Platform FPGA Data Sheet*, Xilinx, Inc, June 2005.