# IMPLEMENTATION OF RUN-TIME RECONFIGURABLE CONSTANT MULTIPLIERS FOR FPGAS

**Dr.SAI**
*Department of ECE, MallaReddy College of Engineering, JNTUH, Hyderabad, India*
*Raydnoces13india@gmail.com*

**Dr.ECCLESTON**
*Department of ECE, MallaReddy College of Engineering, JNTUH, Hyderabad, India*
*pvpathi@gmail.com*

**Dr.THOMAS FELDMAN**
*Department of ECE, MallaReddy College of Engineering, JNTUH, Hyderabad, India*
*Matamshiva1@gmail.com*

**PROF.MUSILEK**
*MallaReddy College of Engineering, JNTUH, Hyderabad, India*
*principal@mrce.in*

## Abstract

This work introduces a new heuristic to generate pipelined run-time reconfigurable constant multipliers for FP-GAs. It produces results close to the optimum. It is based on an optimal algorithm which fuses already optimized pipelined constant multipliers generated by an existing heuristic called RPAG. Switching between different single or multiple constant outputs is realized by the insertion of multiplexers. The heuristic searches for solutions that result in minimal multiplexer over-head. Using the proposed heuristic reduces the run-time of the fusion process, which raises the usability and application domain of the proposed method of run-time reconfiguration. An extensive evaluation of the proposed method confirms a 9-26% FPGA resource reduction on average compared to previous work. For reconfigurable multiple constant multiplication, resource savings of up to 75% can be shown compared to a standard generic LUT multiplier. Two low level optimizations are presented, which further reduce resource consumption and are included into an automatic VHDL code generation based on the FloPoCo library.

**Keywords:** Constant multiplier, runtime reconfiguration, FPGA

## I. Introduct ion

The multiplication with constant coefficients is an essential operation in digital signal processing. Initially one of the rea-sons to put embedded multipliers or DSP blocks into the fabric of field-programmable gate arrays (FPGAs) was to reduce the performance gap between application specific integrated circuits (ASICs) and FPGAs. Nevertheless, the price to pay for those fixed coarse-grained blocks is their inflexibility in word size and limited quantity. Limited quantity is particularly critical in industrial applications, when cheaper and rather small FPGAs with only few DSP blocks have to be chosen due to price pressure. Thus, logic-based constant multiplication methods are needed. Optimizing the implementation of this operation is well studied. Switching between a given set of constants of such multipliers during run-time instead of using larger generic multipliers is important to realize hardware efficient run-time

adaptable filters [1], [2], [3], DCT and FFT implementations [4] as well as multi-stage filters for decimation or interpolation like polyphase FIR filters [5]. A reconfigurable constant multiplier is a multiplication circuit in which the scaling constant can be chosen from a limited predefined set of constants during run-time. For the given application domains two to six of such adjustable coefficient sets are common. The switching during run-time is achieved by inserting multiplexers into several constant multiplication circuits, to achieve a reuse of redundant partial circuits and thus a reduction of required resources. The problem is to find the best possible solution when inserting the fusing multiplexers. In order to avoid large routing delays pipelining

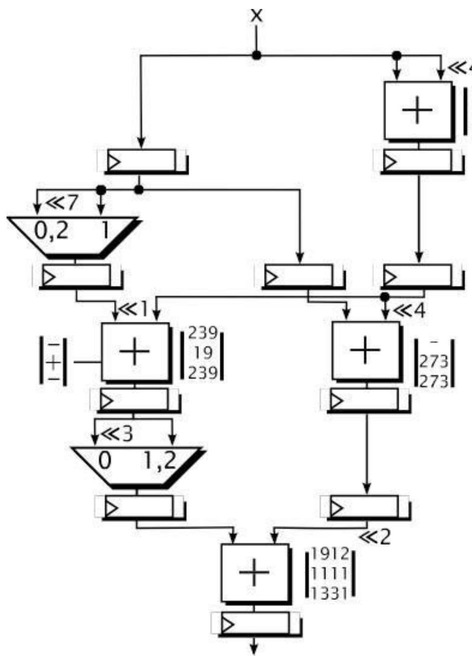


Fig. 1: Reconfigurable single constant multiplier which can be switched between the constants 1912, 1111, 1331.

is used for high speed applications. In contrast to ASICs, this is specifically advisable for FPGA designs [6], due to their inherent performance disadvantage. An example for such a pipelined reconfigurable constant multiplier which can be switched between the constants 1912, 1111, 1331 can be found in Fig. 1. Pipeline registers are inserted after each stage which includes registers in the multiplexer stages. The wires can be associated with a left shift and a sign. The value vector noted besides each operation corresponds to the intermediate or output factors for a specific multiplexer configuration. A switchable adder/subtractor is depicted as an adder with an additional sign vector input.

A common way to realize multiplier-less single and multiple constant multiplication for fixed constants is using additions, subtractions and bit shifts. In general, finding an optimal solution for single constant multiplication (SCM) proved to be NP-complete as shown by Cappello and Steiglitz [7], so optimal solutions can only be found for limited constant bit widths. Optimal SCM solutions for constants of up to 12 bit [8] were first extended to constants of up to 19 bit [9] and further extended for constants of up to 32 bit [10]. Moreover, there are good SCM heuristics called RAG-n, BHM [11] and $H_{cub}$ [12] whose source code as well as an online SCM generator are provided on the SPIRAL project webpage [13]. When FPGAs are the target technology, solutions which consider pipelining during optimization have to be preferred to avoid large routing delays [6]. The problem of finding solutions for pipelined adder graphs (PAGs) for SCM as well as for multiple constant multiplication (MCM), which can directly take advantage of the registers provided in an FPGA's basic logic element, is solved by a heuristic called RPAG [14]. This heuristic was shown to outperform state-of-the-art MCM methods like $H_{cub}$ [12] when these are optimally pipelined [15] and is thus be used as base for the reconfigurable constant multiplication shown in this work. The source code of this heuristic is also available online [16].

Finding the run-time reconfiguration of SCM and MCM adder graphs is a generalization of the basic SCM/MCM prob-lem and thus also NP-complete. However, solutions were pre-sented which are able to find reconfigurable SCMs (RSCM). First of all there are different solutions targeting ASICs, all focusing on multiplexer-based reconfiguration. In the method of Tummeltshammer et al. [17] several optimized SCM graphs are fused by a recursive algorithm called DAG fusion. Two SCM graphs are fused with minimal hardware effort by in-serting multiplexers to switch between the different constants. Further coefficients can be included by recursively adding the related SCMs to the existing RSCM. Similarities between different coefficients in the canonical signed digit (CSD) representation of constants are exploited by Chen et al. [18] to realize RSCMs. Identical patterns in the CSD representation of constants are searched and fused using multiplexers to be able to switch between the different shifts and interconnections to realize a specific constant. Faust et al. [5] use an adder graph based approach with special focus on minimal logic depth. In addition to the methods described earlier, their algorithm does not only provide solutions for RSCM but also for reconfigurable multiple constant multiplication (RMCM). Such RMCM are also provided by ORPHEUS [2] which is able to fuse MCM solutions provided by $H_{cub}$ [12] with a heuristic. Alterative concepts to realize RMCMs are evaluated during the algorithm run-time and the best overall solution is selected. The presented algorithms for RSCM and RMCM, respectively, do not consider pipelining or other FPGA specific issues as their focus is on ASIC implementations.

There is an FPGA-specific algorithm called ReMB method [1] which was further analyzed and extended in [19] by our group. An RSCM is constructed from basic structures that fit into the basic logic elements (BLE) of FPGAs. This procedure is limited to small problem sizes due to a very high memory consumption [19] and does not consider pipelining. As pipelined solutions are required for high speed applications on FPGAs, there is an optimal adder graph based algorithm for RSCM and RMCM with focus on pipelined realizations proposed by our group [20]. The idea of DAG fusion [17] is picked up as already optimized pipelined adder graphs (PAGs) are fused. Instead of fusing only two PAGs in one optimization run, all PAGs of the required constants are considered in one single optimization run to produce a better, multiplexer-aware pipelined realization. However, the optimal approach can only be used for small problems because of the complexity of a full search over all possible fusing solutions. Hence, a good heuristic method is required which provides solutions close to the optimum. This heuristic is presented and analyzed in the following Section II. Moreover, the proposed heuristic supports the use of PAGs consisting of ternary adders [21], [22], [23], which turns out to further improve the results. In Section III, some low level optimizations are provided to exploit the FPGA resources in the best possible way. The results and a comparison with previous work and with the use of generic multipliers are presented in Section IV. A conclusion is given in Section V.

## II. THE PAG FUSION ALGORITHM

### A. *Pipelined Adder Graphs*

The input to the algorithm are pipelined adder graphs (PAGs) generated with the reduced pipelined adder graph (RPAG) heuristic [14]. In general, the presented fusion is not limited to RPAG generated circuits as pipelined MCM input. However, RPAG was chosen as it proved to outperform state-of-the-art MCM methods like $H_{cub}$ [12] when these are optimally pipelined [15]. The results of RPAG are adder graphs representing multiplier-less pipelined constant multipliers us-ing additions, subtractions and bit-shifts only. The main idea of multiplier-less multiplication as applied in RPAG is to compose a constant multiplication of an addition of shifted inputs. This is beneficial because a constant shift is only a wire in hardware. All constants can be formally represented as A-operation [12], which is defined as

$$A_q(u, v) = |2^{l_1} u + (-1)^{sg} 2^{l_2} v| 2^{-r} \quad (1)$$

with $q = (l_1, l_2, r, sg)$, where $u$ and $v$ are the input constants, $l_1$, $l_2$ and $r$ are shift factors and the sign bit $\in sg\{0, 1\}$ de-notes whether an addition or subtraction is performed. A mul-tiplication by 17 could for example be realized as an addition of the input with the input left-shifted by 4 (multiplication by 16). This can be seen in the leftmost example in Fig. 2. In the following subtractor, 17 times the input is subtracted from 256 times the input, which corresponds to a constant multiplication by 239. Finally, this intermediate result is left-shifted by three to get the final result of 1912 times the input. If the constant to multiply with is known in advance, this kind of realization is much cheaper in terms of resources than implementing a generic multiplier [15]. In order to automatically generate such constant multipliers, RPAG is backward-exploring reachable intermediate constants, called predecessors by evaluating the A-operation. This leads to a step-wise constant composition, starting with the required output constants. The goal of the heuristic is to select predecessors which result in the lowest number of intermediate constants in the preceding stage and which reduce the adder depth. Two more examples for such a circuit of a pipelined SCM realization can be found in Fig. 2, which are used as running example. The stage $s$ denotes the pipeline depth of each realized constant.

### B. *Improved Pipelined Adder Graph Fusion*

Just like RPAG, the proposed pipelined adder graph fusion is backward-exploring. Starting with the constant mapping of the output stage all PAGs are fused stage by stage. The basic idea is to combine those intermediate values in the respective preceding stage to share the same adder, which leads

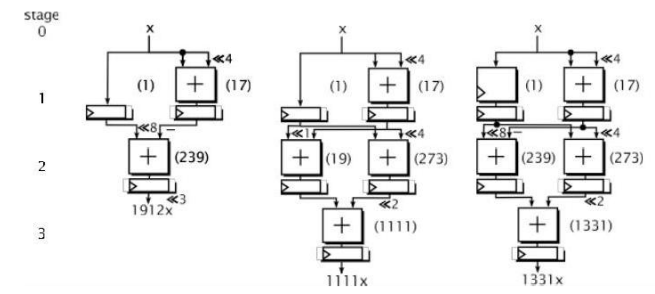


Fig. 2: RPAG solutions for the constants 1912, 1111, 1331.

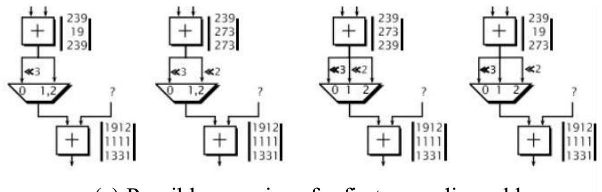
to a minimal overhead of possibly necessary multiplexers or switchable adder/subtractors. To do so, all combinations of intermediate values are evaluated and their costs are calculated separately and stored in a cost matrix. Multiplexers can appear at the inputs of the successive stage in the following cases:

1) input has a different shift value
2) input has a different source
3) both of 1) and 2)

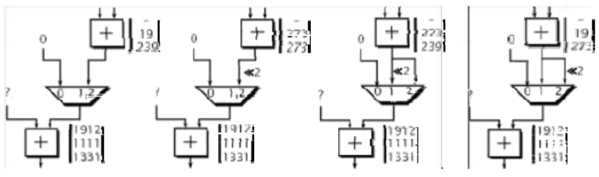
As described before, the target is to select the overall best mapping *M* for the specific stage *s*. This selection will be the source for the determination of the next preceding stage $s-1$. The procedure is repeated until the input (stage 0) is reached. A simplified pseudo-code of the generalized fusion process is given in Listing 1. It assumes that the overall best solution and costs are globally known. It is started with the constant mapping M of the output stage, the preceding stage s, the search width w (unlimited for the optimal search) and the costs of the current path *current cost,* which is zero in the beginning. Compared to the algorithm presented in [20] the algorithm was generalized, such that it can be used both as heuristic and in an optimal way. In contrast to an arbitrary search through the whole search space, which was done in the former version, the search is now improved and based on a sorted cost matrix. More details on the search width are provided in Section II-D.

Listing 1: Simplified overview of the main recursion of the improved fusion algorithm with desired output mapping *M* of current stage, preceding stage *s*, the search width *w* and cost of the current path *current cost*.

```
1  Fuse (M,s,w,current_cost)
2  if s > 0
3    C = evaluate_fusion_cost(M);
4    C = sort(C);
5    if current_cost+min(C)>=current_best_cost
6      return; -- subtree cut
7    else
8      for i = 0...w
9        if current_cost+cost(C(i))>=current_best_cost
10         return; -- normal b&b cut
11       else
12         M = mapping(C(i));
13         current_cost += cost(C(i));
14         Fuse (M,s-1,w,current_cost);
15       end if;
16     end for;
17   end if;
18 else
19   if current_cost < current_best_cost
20     current_best_cost = current_cost;
21     best_solution = current_solution;
22   end if
23 end if;
```



(a) Possible mappings for first preceding adder.



(b) Possible mappings for second preceding adder.

Fig. 3: All combinations of the adders in the second last stage for the given output mapping and the given RPAG SCM solutions in Fig. 2.

In the running example used here, the three SCM graphs generated with RPAG (see Fig. 2) are fused starting with the desired output mapping $M = 1912; \{1111; 1331\}$ , meaning that the resulting circuit can be switched between these three values. This will be called an SCM circuit with three *configurations* in the following.

The enumeration of all adder combinations of the second last stage consisting of$\{239\}$ for the first,$\{19\}\{,273\}$ for the second and $\{239\}$ ,$\{273\}$ for the third SCM solution, respectively, is given in Fig. 3. For the constant 1912 only one adder is required in stage two, but two adders are required in the other SCM circuits. This fact is considered by the insertion of a don't care "− ". Due to a separate cost calculation for a specific combination, some of the multiplexer inputs are unknown from a local point of view. These are marked with a question mark. They do not have any contribution to the currently considered adders' multiplexer costs, which is a main advantage of the proposed method as the costs for each combination can be calculated and evaluated separately.

The cost evaluation is following the assumption that the multiplexers will be realized as a cascade of 2:1 multiplexers. Thus, $N-1$ 2:1 multiplexers are required to switch between $N$ configurations, which leads to a contribution of each used multiplexer input of:

$$\text{cost}_{\text{MUX}} = \frac{N-1}{N} \tag{2}$$

As a zero input can be realized by resetting the succeeding register, these inputs are not considered as multiplexer inputs, as our implementation targets pipelined implementations. The multiplexer cost for each mapping is stored in a multidimensional cost matrix *C* (line 3 in Listing 1). The cost matrix for the combinations of the current stage can be found in TABLE I in a two dimensional representation. For example, the first entry in the first row (1.33) corresponds to the leftmost mapping in Fig. 3 (a) in which two multiplexer inputs, each with a cost$_{\text{MUX}}$ of $\frac{2}{3}$ are used.

To get a valid solution a selection of one mapping for the first preceding adder in Fig. 3 (a) will directly force the

TABLE I: Cost matrix for stage 2 fusion of the given example.

| | 19, 239 | 273, 273 | 273, 239 | 19, 273 |
|---|---|---|---|---|
| 239 | **1.33** | 1.33 | 2 | 2 |
| – | 0.67 | **0.67** | 1.33 | 1.33 |

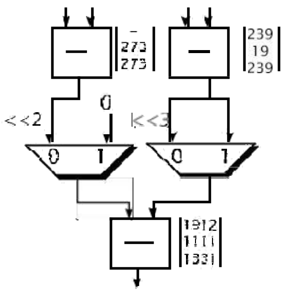


Fig. 4: Result of combining 239,19,239 and– ,273,273 as preceding adders.

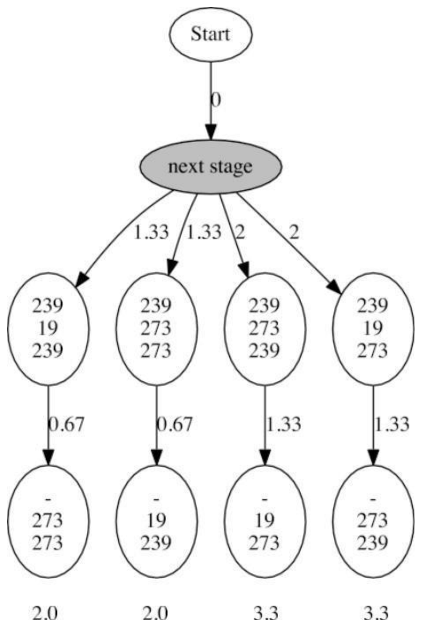


Fig. 5: Part of the decision tree for the second stage fusion.

selection of the corresponding mapping for the second adder (Fig. 3 (b)) or reduces the selectable possibilities for other adders in a more general case. This means each valid mapping solution for a specific stage consists of selections with a unique row and column index. Thus, finding the cheapest mapping solution *M* for a specific stage reduces to finding the valid solution with the lowest sum of costs. An example for such a selection is given in Fig. 4. It corresponds to the highlighted selection in TABLE I with a total cost contribution of 1.33 + 0.67 = 2 2:1 multiplexers.

The cheapest solution for a specific stage is not necessarily the best overall choice as it affects the costs in the preceding stages. So, to find the optimal solution, a full search over all possibilities is necessary. The search space can be illustrated as a decision tree, which consists of the decision itself as node and the cost of the decision as edge. An example for this can be found in Fig. 5, which shows a decision tree representation for the second stage fusion decisions of TABLE I.

Each branch of the tree is a valid mapping solution for the specific stage which can be chosen as output mapping for the preceding stage (line 12 in Listing 1). Each branch corresponds to a recursive call of *Fuse()* in line 14 of Listing 1. Thus, to get the full search space, a different decision tree for the preceding stage has to be added recursively for each stage's output mapping until the input stage is reached. As the cost matrix *C* is sorted in line 4 of Listing 1 the algorithm follows the best cost solutions first. In the case of equal cost solutions, the first solution found with these costs, is chosen as the cheapest one. The full decision tree for our example can be found in Fig. 6 and can be generated by setting *w* to $\infty$ .

As shown in the complexity analysis in the next section, the full search for the optimal solution can be very time consuming for larger problems as the number of combinations grows factorial with the number of adders $K_s$ in one stage and exponential with the number of configurations *N* . Nevertheless, the memory consumption is moderate as for the presented algorithm, as only the local environment has to be stored, which is the currently optimized branch. Moreover, irrelevant branches, which are too expensive anyway, can be pruned whenever the currently best cost value is exceeded. This is why the search is started with the locally best solution, which already finds good solutions in the first iterations. It is executed in a branch-and-bound way, which stops searching a branch if the total costs exceed the current global minimum costs. Besides the normal branch-and-bound method (cf. Listing 1, line 9-10), denoted as *cut* in the decision tree in Fig. 7, an additional pruning criterion is added (cf. Listing 1, line 5-6). The sum of minimum values of each row in the cost table is a lower bound of costs which can be added by the considered stage. If this minimum added to the current costs is larger than the global minimum, the whole subtree can be pruned, denoted as *stcut*. The best resulting reconfigurable single constant multiplier solution of the running example is shown in Fig. 1. It corresponds to the leftmost branch in the decision tree of Fig. 6 and Fig. 7.

### C. *Complexity Consideration: Analysis of the Search Space*

As the presented optimal PAG Fusion algorithm has to traverse the full search space, its complexity, i.e., the number of possible solutions and branches to find them, is an important issue.

*1) Number of Solutions:* One key measure is the number of possible solutions to combine the adders in one stage. *N* denotes the number of configurations while the number of adders in the considered stage *s* of the input adder trees is $K_s$. The total number of adder combinations which are possible is $(K_s!)^N$ . As the arrangement of the combinations itself does not matter, it can be ignored during the enumeration of solutions. As there are $K_s$! ways to arrange the combinations,

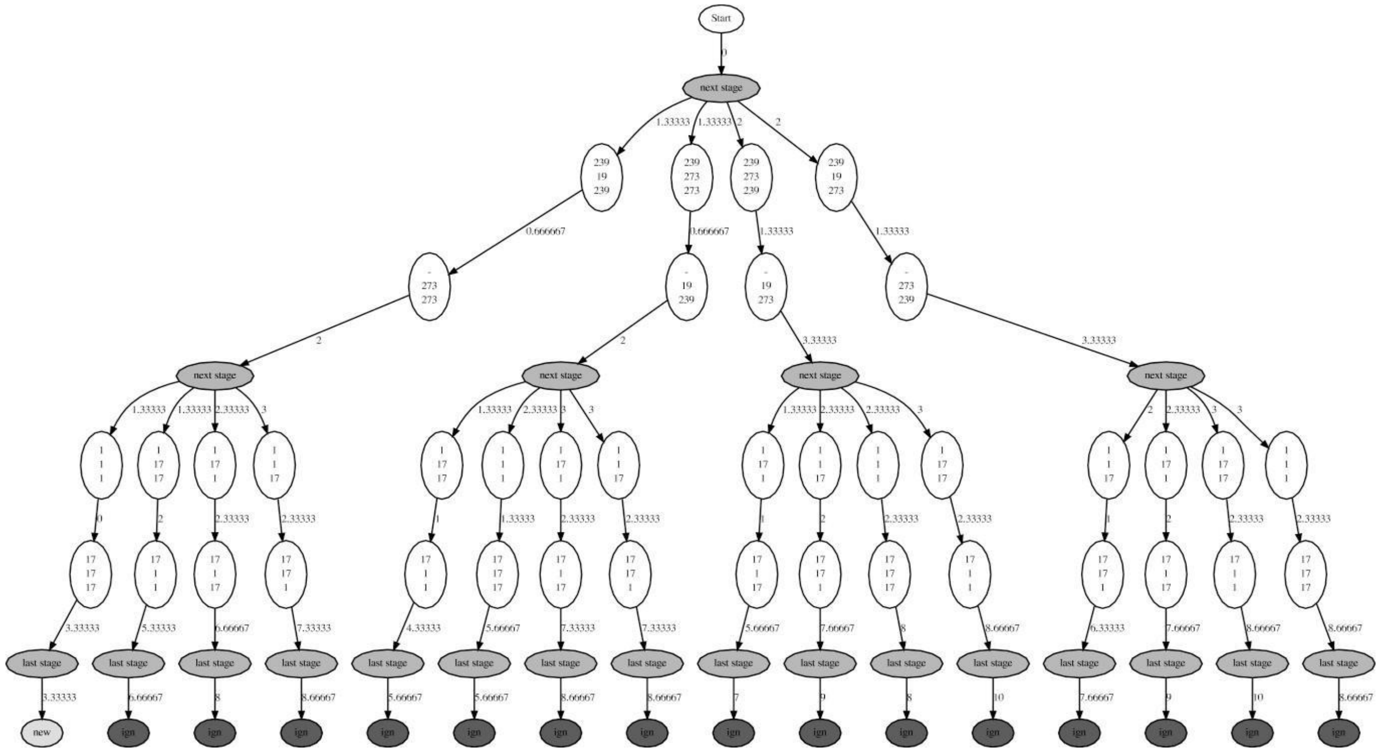


Fig. 6: Full decision tree for the example with costs for each decision, best total costs (light gray) and worse total costs (dark gray).

the total number of solutions $L_s$ for a specific stage's $s$ subtree is

$$L_s = \frac{(K_s!)^N}{K_s!} = (K_s!)^{N-1} \tag{3}$$

In the example decision tree of Fig. 6, $L_1 = L_2 = 2^{(3-1)} = 4$, which corresponds to the gray nodes in each subtree, as we have two nodes to fuse and three configurations in each stage.

*2) Number of Decisions:* In order to evaluate the run-time of the algorithm the number of decisions $D$, which is equal to the number of nodes in the decision tree, is important. This number depends on the possible solutions $M_s$ for the subtrees and the possibilities to reach them. For one stage's $s$ subtree the number of decisions is calculated as

$$D_s = L_s \cdot \sum_{j=0}^{K_s-1} \frac{1}{j!} \cdot \underbrace{\sum_{j=0}^{N-1} \frac{1}{j!}}_{\le e \text{ (Euler Number)}} \tag{4}$$

In the example decision tree of Fig. 6, $D_1 = D_2 = 4(1^{(3-1)} + 1^{(3-1)}) = 8$, which corresponds to the white nodes for each subtree. Note that, although the sum consists of a factorial and an exponential part, it has an upper limit of $e$ (Euler Number).

For the whole decision tree the total number of decisions is the sum of decisions in each stage. This is the number of

This means for our running example that though we have only 16 possible solutions, 40 (= 8·1+8·4·1) decisions are required to build them.

The equation for the total number of decisions (5) shows

that the search space to find the optimal solution grows at least factorial with the number of adders per stage $K$ and exponential with the number of configurations $N$, as it contains a product term of the $L_s$ (cf. (3)). For larger benchmarks this is not applicable, not even when branch-and-bound is applied. Therefore a heuristic is needed, to reduce the considered search space.

## *D. PAG Fusion for Larger Problems*

The presented PAG fusion approach is able to find valid solutions for pipelined RSCM and RMCM adder graphs. Finding the optimal solution can not be guaranteed for large

problems, which can lead to drastic time consumption. Thus, a heuristic to find a close-to-optimal solution with controllable time consumption is required. Finding a good heuristic is not trivial, as it is not clear which strategy is appropriate for the present search space. An analysis of the search space of PAG fusion showed, that selecting branches with low costs in the local decision phase, raises the likelihood to find optimal or Close-to-optimal solutions. Thus, the heuristic presented here works by limiting the number of branches to the ones which of the preceding stages) multiplied by the number of subtree decisions:

$$D = \sum_{i=1}^{S-1} D_i \prod_{j=1}^{i-1} L_j \qquad (5)$$

$i=1 \quad j=1$

are most likely to be included in the
are the cheapest solutions in each search tree stage. The search branches are limited by the so called search width $w_{search}$.The value of $w_{search}$ specifies the number of additional branches

value of $w_{search}$ specifies the number of additional branches

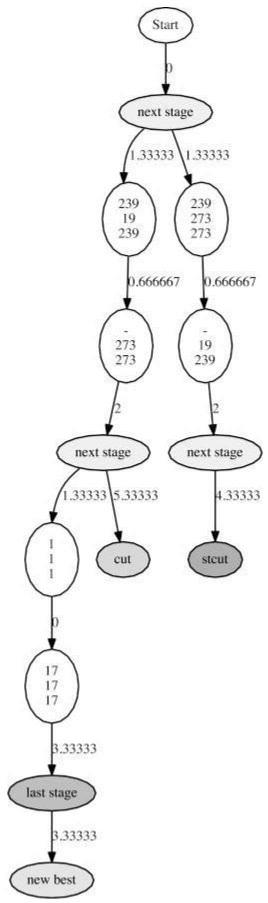


Fig. 7: Reduced search space of $w_{search}$ = 1 for the given example

which are searched with the locally best solution. Thus, the search strategy is a modified breadth-first search and is related to the *beam search* strategy [24]. Within the meaning of this definition the initial greedy search of the algorithm has a search width of $w_{search}$ = 0. As described earlier in Section II-B the decision for the next path to follow in the decision tree is made by using a cost matrix for each considered subtree in each stage. For the heuristic these cost matrices are sorted (cf. line 4 in Listing 1), to be able to quickly access the $w_{search}$ + 1 best paths. When a path is selected, this selection equals the desired output mapping for the previous stage. For that previous stage the cost matrix is evaluated in the same way. This provides a very plain and fast method to reduce the search space. Note that the described branch -and-bound cuts can also be applied within the heuristic, which has the potential to further speed up the search. Especially the subtree cut (stcut), which was already explained in Section II-B is now very easy to perform due to the sorted use of the cost matrix within the heuristic. An example for the search space of $w_{search}$ = 1 can be found in Fig. 7. The branch-and-bound cuts are denoted as *cut*, while the subtree cuts are denoted as *stcut*. Only eight instead of 40 decisions are needed compared to the full search for this rather small example to find a good –in this case the best– solution. This heuristic provides an easy way to directly control the run-time and offers the possibility to realize larger switchable coefficient sets for single and multiple constant multiplication.

## E. *Exploiting Ternary Adders*

Adders with three inputs (ternary adders) can significantly reduce the number of operations in a pipelined adder graph generated by the RPAG heuristic [21] and thus, reduce the required hardware. Support for ternary adders is given in recent Altera and Xilinx FPGAs, namely Arria I, II, V, 10, Stratix II-V, 10 and Virtex 5-7 [22], [23]. That is why the fusion of such PAGs using three-input adders was also integrated into PAG fusion, taking the implementation available at OpenCores [25]. The target is to reduce the number of operations and with it the required multiplexer inputs. Instead of the two operations $a + b$ and $a - b$ each adder in the adder graph is now able to implement $a + b + c$,$-a\ b + c$, $a +- b\ c$ or $-a - b\ c$. The cost evaluation and the decision for a specific stage's mapping was adapted to this circumstance. This includes an extension of the data structure to provide nodes with three inputs. Special care has to be taken, when nodes with two and nodes with three inputs are fused. Here an addition with 0 has to be provided for the input, which is unused in one circuit. This leads to an additional degree of freedom, when selecting the node's input mapping and evaluating a mapping's costs. Another extension had to be provided in the fusion of subtractors. In the two-input adder case it is possible to map all negative inputs to the same input resulting in a subtractor instead of a switchable adder-subtractor. This is not always possible in the three input adder case ($a - b - c$). However, the swapping of inputs to get the best possible solution was adapted for the ternary adder cases ($a - b + c$, $a + b - c$ ), in which a swapping can help to reduce the required resources. All these adaptions are leading to a larger complexity for the consideration of inputs in these steps. Nevertheless, the overall run-time is supposed to be reduced, as the number of adders $K_S$ per stage is reduced due to the operation reduction shown by Kumm et al. [21].

## III. Low Level Optimizations

### A. *Multiplexer Mapping*

As multiplexers are used to switch between the different constants, their mapping to the target FPGA should be as good as possible. This can be achieved by using the explicit resources provided in Xilinx Virtex 5-7 slices [26]. In the case of the used Virtex 6 FPGA, our VHDL code generator produces the optimal mapping using *Primitives* [27]. This results in a resource optimized multiplexer implementation. The gain of this implementation can be seen in Fig. 8, which shows the LUT consumption of the mapping achieved by Xilinx ISE 13.4 (gray) as well as the improved solution by using *Primitives* and the resource optimal mapping [26] (black when better, otherwise equal to ISE mapping). The operating frequency is not shown, due to the fact that only one up to three slices are required, which leads to frequency estimations between 770 and 1.300 MHz, which is unrealistic for a final design, as there should be more limiting parts elsewhere. In 16 of the 31 cases one LUT per bit can be saved. That is why the inclusion of this mapping as operator into our FloPoCo-based [28] VHDL generator is part of this work.

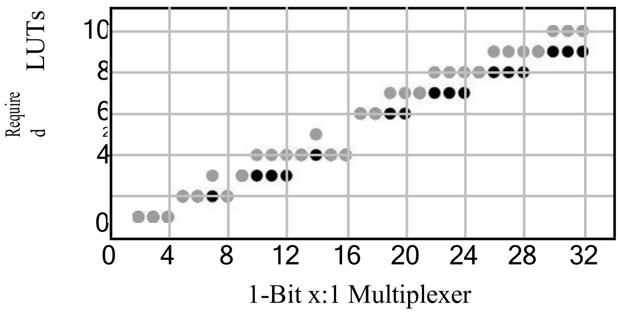


Fig. 8: Required LUTs for 1-bit x:1 multiplexer. ISE solution (gray) and improvement by *Primitive* usage [26] (black).

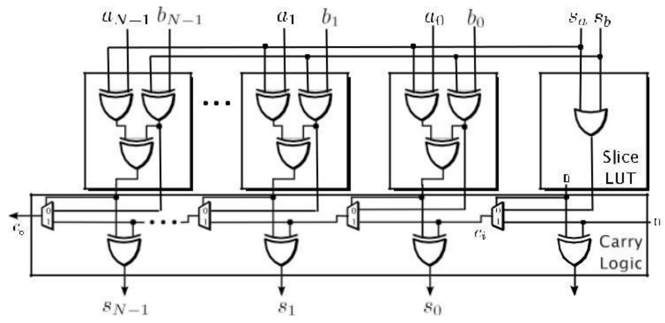


Fig. 9: Realization of switchable adder/subtractor on Xilinx Virtex 5-7 slices

*B. Switchable Adder Subtractor Mapping*

The fusion of adders with subtractors leads to the re-quirement of switchable adder/subtractors in which the input that is subtracted can be either input *a* or input *b*. The proposed realization of the switchable adder/subtractors on Xilinx Virtex 5-7 slices can be found in Fig. 9. The realization is done using a single LUT to provide the correct carry input and the following LUTs to provide an XOR of the inverted or non-inverted inputs, which builds a full adder together with the slice's carry logic. The inputs are inverted when required by an additional XOR of each input with the corresponding subtraction flag $s_a$ or $s_b$. The subtraction flag $s_a$ or alternatively $s_b$ has to be set to 1 if *a* or *b*, respectively, should be subtracted. The case in which $s_a$ and $s_b$ are both 1 is not supported by the given implementation. Using the described switchable adder/subtractor together with the opti-mized multiplexer implementation helps to further reduce the required slice resources. In our experiments we observed cases, in which the more general VHDL implementation needed more than twice as many LUTs. So when a switchable adder subtractor is required more than 50% of slice resources can be saved in the best case with the proposed implementation.

## IV. Results

This section provides synthesis results to highlight the advantages of the proposed method. For all experiments the same VHDL code generator which is based on the FloPoCo library [28] was used to create synthesizable VHDL code. VHDL code was generated with identical settings and mapped to a Virtex 6 FPGA (xc6vlx75t-2ff484-2) using Xilinx ISE

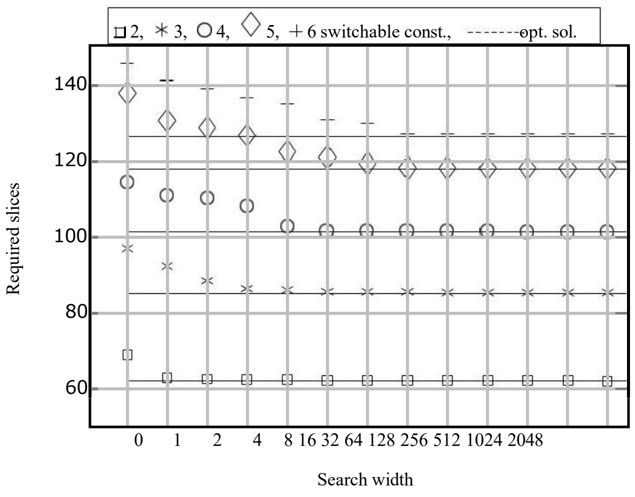


Fig. 10: Comparison of the required slices of the heuristic with different search widths and optimal solution (dashed lines).

TABLE II: Comparison of the average run-time of the heuristic to the optimal method and average area overhead of the heuristic solution

| | run-time [s] | | speedup | area degradation |
|---|---|---|---|---|
| | heur. $w = 64$ | opt. [s] | heuristic | heuristic |
| 2 conf. | <0.001 | <0.001 | 1.00 x | 0.41% |
| 3 conf. | 0.00188 | 0.00190 | 1.01 x | 0.69% |
| 4 conf. | 0.9293 | 0.9355 | 1.01 x | 0.37% |
| 5 conf. | 2.51 | 120.52 | 4.80 x | 0.46% |
| 6 conf. | 14.16 | 1842.24 | 130.10 x | 0.96% |

v13.4. The proposed algorithms' input graphs to be fused were created using the RPAG heuristic. The source code of RPAG and of the proposed method is available online as open source within the PAGSuite [16].

*A. Quality of the Heuristic*

In order to evaluate the quality of the heuristic, an SCM benchmark for 2 to 14 configurations, each consisting of 100 constant sets using randomly generated constants uniformly distributed between 1 and $2^{16}-1$ was created. In this ex-periment, optimal solutions were generated to have a baseline for the heuristic results. In addition to that, solutions using the heuristic with different search widths for all benchmark sets were generated. A direct comparison of the average slice utilization of the optimal PAG fusion and the heuristic for constant sets with 2 to 6 configurations can be found in Fig. 10. It shows the required slices for the different search widths and the optimal solutions with dashed lines. It can be observed that a search width of only 64 leads to solutions close to the optimal solution for the RSCM benchmark sets with up to 6 configurations. The maximum operating frequency of all solutions is distributed equally between 443 and 469 MHz. The average run-time for a width of 64 in this experiment is compared in TABLE II. It can be observed that a longer run-time of the optimal algorithm leads to an increased speedup for the heuristic. At the same time, the area degradation of the heuristic is smaller than 1%. This encourages the use of the heuristic for larger numbers of outputs as required for RMCM problems, which have a much larger complexity due to a larger adder count per stage.

## *B. Comparison to DAG Fusion*

This section shows a comparison of the proposed algo-rithm to the DAG fusion algorithm [17] which also relies on the fusion of adder graphs. The same benchmark as for the heuristic classification was used. Using this benchmark pipelined adder graphs with the proposed PAG fusion heuristic as well as pipelined and non-pipelined adder graphs with DAG fusion using the available SPIRAL source code [13] were generated. The pipelined DAG fusion results were obtained by inserting registers after each adder, subtractor, adder/-subtractor, multiplexer and corresponding pipeline balancing registers. Pipelined results for DAG fusion are needed for a fair comparison of the slice utilization and performance evaluation. The proposed algorithm was executed with a search width of 64 as motivated in the last section. DAG fusion was executed with a restricted mode provided in the DAG fusion code when the run-time exceeded 3 hours (typically needed for cases with more than 9 configurations). The results for the required slices after place and route and the maximum clock frequency can be found in Fig. 11. Note that each data point is an average value of 100 constant sets. As a baseline, a 16× 16 bit CoreGen [29] soft-core multiplier (LUT-based implementation) with the same pipeline depth as our solutions together with distributed RAM to store the coefficients is shown in Fig. 11. For the pipelined implementations it can be observed that the proposed algorithm has a lower slice utilization than DAG fusion in all cases. Compared to DAG fusion, the proposed method provides a slice reduction of 9% on average when 2-input adders are considered and 26% on average when ternary adders are considered. The resulting 2-input adder circuits can be run at nearly the same maximum clock frequency as the pipelined DAG fusion circuits and the CoreGen reference. Due to pipelining, the proposed method and pipelined DAG fusion have a similar critical path, which can be found in the adders or in the multiplexers with varying size. For the ternary adders there is a performance degradation of about 39% on average which was also reported by Kumm et al. [21]. The non-pipelined DAG fusion results are in some cases better than the pipelined 2-input and ternary adder results, but the maximum clock frequency is up to 5 times slower. This clearly shows the necessity of pipelining on FPGAs. The comparison between the DAG fusion results and the results of the proposed method also show that an optimization which considers all configurations in a single run leads to better results. In general, it can be seen that the proposed method is valuable for up to four configurations in the 2-input adder case and up to six configurations in the ternary adder case, when the required slices are considered. For more configurations, the soft-core multiplier implementation by CoreGen provides the solution which requires the least resources. For ASICs, DAG fusion proved to be valuable for up to 19 coefficient RSCM (cf. Table II in [17]). This appears to be a maximum gap between the optimized adder implementation and a generic multiplier.

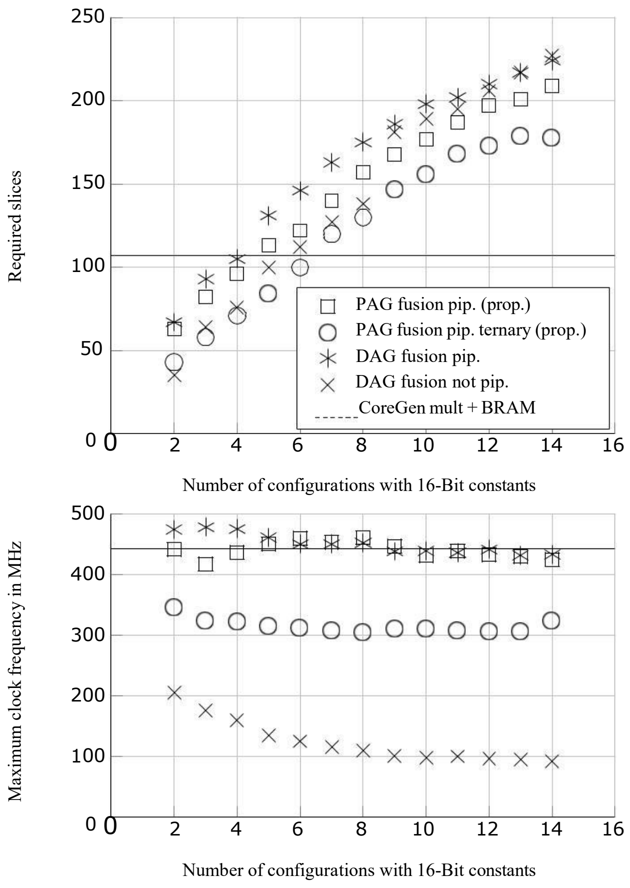


Fig. 11: Comparison of the required slices (top) and the maximum clock frequency (bottom) for the proposed method and DAG fusion [17].

This has to be of course smaller for FPGAs. For this small gap, which is a relevant field for many applications, the pro-posed heuristic can generate solutions with significantly lower resource consumption and similar performance. When only RSCM is considered, optimal solutions are possible, when about half an hour of run-time is feasible. But the heuristic is unconditionally required to enable shift-add-based reconfig-urable multiple constant multiplication, especially with many outputs, which is required, e.g., for run-time reconfigurable FIR filters.

## *C. Reconfigurable Multiple Constant Multiplication*

When reconfigurable multiple constant multiplication is considered, it can again be compared to the CoreGen soft-core multiplier with RAM for the coefficients. To have more than one output, multiple CoreGen multipliers and coefficient RAMs are used. A benchmark for 5 different MCM cases (2, 4, 6, 8 and 10 outputs), each with 2, 4, 6, 8 and 10 configurations, consisting of 50 constant sets per case using randomly generated constants uniformly distributed between 1 and $2^{16}-1$ was created. The search width was again set to 64. The results of the 2-input and 3-input adder implementations compared to CoreGen multipliers can be found in Fig. 12.

It can be seen that the CoreGen soft-core multiplier imple-mentation is better for 6 or more configurations in the 2-input

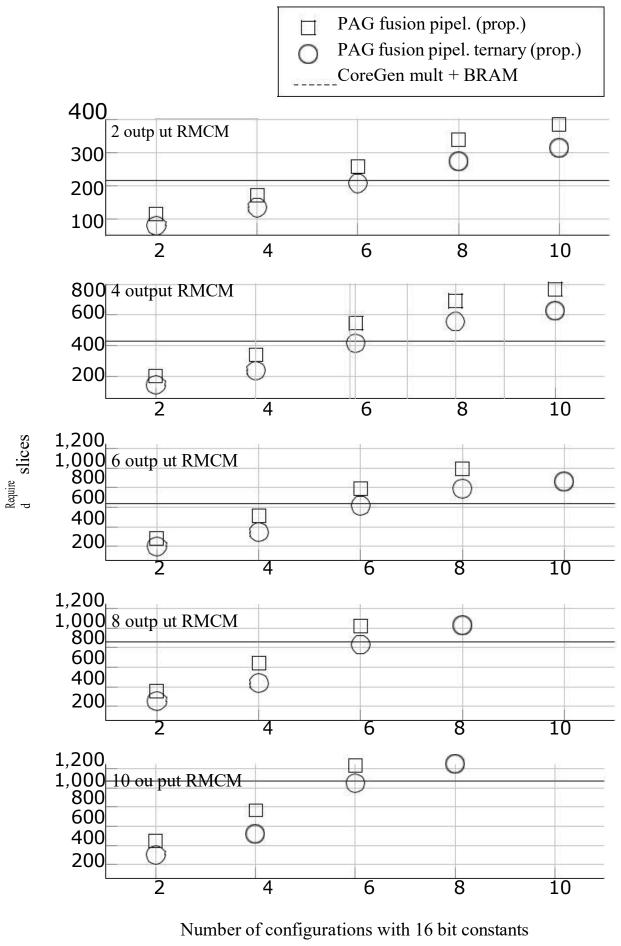


Fig. 12: Comparison of RMCM and tRMCM to a CoreGen soft-core multiplier + RAM

adder case and 8 or more configurations in the ternary adder case. Below these numbers of configurations, up to 75% of the resources can be saved, when the proposed reconfigurable shift and add based implementation is preferred, which is up to 750 slices in the 10 output RMCM case. Note that without the heuristic only the results for 2 outputs and 2 configurations could have been generated optimally within a run-time limit of 3 hours. MCM solutions normally have more adders in each stage, which leads to a much larger search space and thus a much larger run-time. Using the heuristic with its controllable search width, raises the solvable problem size and thereby enables the application domain of RMCM for the proposed fusion algorithm. For the application domains given in the introduction [1], [2], [3], [4], [5], 2 to 6 MCM configurations are common, which is the range of the proposed heuristic. Up to 75% of slice resources can be saved compared to a generic multiplier.

## D. Comparison to Other Reconfiguration Approaches

If the presented multiplexer-based switchable multiple constant multiplication is compared in the context of reconfigurable circuits, the reconfiguration time is an important factor. The presented approach has a reconfiguration time of only one clock cycle which is about 2-3 ns for the mapped and routed designs. To compare our method to other reconfiguration approaches, a 41 tap benchmark FIR filter (MIRZAEI10 41_ [30]) was used. This benchmark was already used in prior work [31], to compare internal configuration access port (ICAP) reconfiguration of Xilinx FPGAs and two logic based reconfiguration approaches. In the benchmark the original filter was extended to a run-time reconfigurable FIR filter by designing additional different FIR filters with the same length as the original benchmark filter and an input word size of 16 bit. These were optimized by using RPAG [14] and can be reconfigured via ICAP. Alternatively, the benchmark set was realized using the two logic based reconfiguration approaches (FIR DA and FIR LUT). In the FIR DA approach the FIR filters were realized using Distributed Arithmetic [32] with a LUT-based implementation and were made reconfigurable by using run-time configurable 5-input LUTs in Xilinx Vir-tex FPGAs. The FIR LUT approach is based on the KCM method [33], in which a constant multiplier is built by several smaller LUT multipliers, whose shifted outputs are finally added. Reconfiguration was again achieved by using run-time configurable 5-input LUTs in Xilinx Virtex FPGAs. These have a configuration time of 32 clock cycles, leading to a reconfiguration time of about 61 to 66 ns in the analyzed filters. The presented PAG fusion heuristic can be used to generate the multiplication of switchable filter coefficients (RMCM) with the input of a transposed FIR filter, which are then followed by structural adders. The results for the previous work and results for PAG fusion RMCM FIR filters for 2 to 5 configurations (conf.) are listed in TABLE III. In this case the number of configurations corresponds to the number of different FIR filter coefficient sets. The ICAP resource consumption and maximum clock frequency are noted as a range as the applied RPAG optimization heavily depends on the numeric coefficient values. In addition, a FIR filter using CoreGen multipliers together with RAM was evaluated. It can be seen that the resulting circuits of the proposed method provide the fastest reconfiguration time with a better resource consumption for 2 to 4 configurations. The large increase in slices from 4 to 5 configurations can be directly traced back to the increase of LUT costs for the 5-input multiplexers (cf. Fig. 8). For 5 to 10 configurations it depends on the required reconfiguration time, if the reconfigurable FIR filter using distributed arithmetic or LUT multipliers together with reconfigurable LUTs or the ICAP implementation should be used. An implementation with CoreGen multipliers is only valuable when very fast reconfiguration times and at the same time a large number of configurations are required.

## E. DSP Block Usage Considerations

On modern FPGAs, DSP blocks in combination with RAM for the coefficients can be used instead of the proposed run-time reconfigurable constant multiplication. If limited quantity of DSP blocks is not a problem, each of the 16 x16 bit multipli-ers of reported cases could be replaced by one DSP block and two slices for the coefficient RAM. For multiplication word sizes larger than 18 Bit, more than one DSP block would be required for Xilinx FPGAs. A comparison of the usage

TABLE III: Comparison of a single filter MIRZAEI10 41 with $B_x$ = 16 bit using ICAP reconfiguration, CFGLUT methods, the proposed PAG Fusion heuristic and CoreGen mutlipliers.

| Method | $S$ [bit] | Slices | $f_{\text{clk}}$ [MHz] | $T_{\text{rec}}$ [ns] |
|---|---|---|---|---|
| RPAG [34] with ICAP | 746496 | 502. . . 569 | 386.7. . . 448.8 | 233280 |
| Reconf. FIR DA [35] | 1920 | 1071 | 521.9 | 61.3 |
| Reconf. FIR LUT [31] | 14784 | 1108 | 487.8 | 65.6 |
| PAG Fusion (2 conf.) | 0 | 848 | 401.3 | 2.5 |
| PAG Fusion (3 conf.) | 0 | 911 | 372.2 | 2.7 |
| PAG Fusion (4 conf.) | 0 | 968 | 402.7 | 2.5 |
| PAG Fusion (5 conf.) | 0 | 1590 | 340.0 | 2.9 |
| CoreGen mult | 3360 | 2647 | 343.9 | 2.9 |

of DSP blocks to the proposed slice based method can be done by relating the two types of resources (DSP blocks and slices) according to their relative availability, referencing their utilization ratio [15]. Alternatively, the chip area consumed by the resources can be related [36]. However, neither of the two methods addresses the frequent requirement to select the smallest, hence cheapest, possible FPGA the design fits into. Usually, in a complete design other parts are competing for DSP resources in digital signal processing applications [37], [38]. For such cases a trade-off must be available. This is provided by the proposed slice based run-time reconfigurable constant multiplier implementation.

## V. Conclusion

This work presented a new heuristic to generate pipelined run-time reconfigurable constant multipliers based on an optimal algorithm. The heuristic was motivated by a complexity consideration of the search space. With the heuristic problems with a larger size become solvable. An extensive benchmark evaluation showed superiority over previous work, as we could show a 9-26% slice reduction on average. Additional extensions to the algorithm were presented which further reduce the slice consumption of the resulting solutions. These were the support of ternary adders, and optimized multiplexer and switchable adder/subtractor mapping. Finally it could be shown by RMCM and FIR filter experiments that the heuristic is raising the solvable problem size and the application domain of the proposed fusion method. Compared to other reconfiguration approaches our method provides the fastest reconfiguration time with a low resource consumption for a limited number of configurations. The source code of the proposed method is available online as open source within the PAGSuite project [16] to increase reproducibility and provide it for future research.

## Acknowledgement

The authors would like to acknowledge the work of Dipl.-Math. Evelyn Lerche from the University of Kassel who provided some nice hints for the complexity considerations.

## References

[1] S. S. Demirsoy, I. Kale, and A. G. Dempster, “Reconfigurable Multiplier Blocks: Structures, Algorithm and Applications,” *Circuits, Systems and Signal Processing*, vol. 26, no. 6, pp. 793–827, 2007.

[2] L. Aksoy, P. Flores, and J. Monteiro, “Multiplierless Design of Folded DSP Blocks,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 20, no. 1, pp. 1–24, Nov. 2014.

[3] P. Lowenborg and H. Johansson, “Minimax Design of Adjustable-Bandwidth Linear-Phase FIR Filters,” *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 53, no. 2, 2006.

[4] M. Garrido, F. Qureshi, and O. Gustafsson, “Low-Complexity Multiplierless Constant Rotators Based on Combined Coefficient Selection and Shift-and-Add Implementation (CCSSI),” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 7, pp. 2002–2012, July 2014.

[5] M. Faust, O. Gustafsson, and C.-H. Chang, “Reconfigurable Multiple Constant Multiplication Using Minimum Adder Depth,” in *Signals, Sys-tems and Computers, Conference Record of the Forty Fourth Asilomar Conference on*, Nov 2010, pp. 1297–1301.

[6] U. Meyer-Baese, J. Chen, C. H. Chang, and A. G. Dempster, “A Comparison of Pipelined RAG-n and DA FPGA-based Multiplierless Filters,” in *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, Dec 2006, pp. 1555–1558.

[7] P. R. Cappello and K. Steiglitz, “Some Complexity Issues in Digital Signal Processing,” *Acoustics*, vol. 32, no. 5, pp. 1037–1041, Oct. 1984.

[8] A. G. Dempster and M. D. Macleod, “Constant Integer Multiplication Using Minimum Adders,” *Circuits, Devices and Systems, IEE Proceedings*, vol. 141, no. 5, pp. 407–413, Oct 1994.

[9] O. Gustafsson, A. G. Dempster, and L. Wanhammar, “Extended Results for Minimum-Adder Constant Integer Multipliers,” in *Circuits and Systems (ISCAS). IEEE International Symposium on*, vol. 1, 2002, pp. I–73–I–76.

[10] J. Thong and N. Nicolici, “A Novel Optimal Single Constant Multiplication Algorithm,” in *Design Automation Conference (DAC), 47th ACM/IEEE*, June 2010, pp. 613–616.

[11] D. R. Bull and D. H. Horrocks, “Primitive Operator Digital Filters,” *Circuits, Devices and Systems, IEE Proceedings*, vol. 138, no. 3, pp. 401–412, Jun 1991.

[12] Y. Voronenko and M. Puschel, “Multiplierless Multiple Constant Multiplication,” *Transactions on Algorithms (TALG)*, vol. 3, no. 2, May 2007.

[13] SPIRAL-Project. (2016) http://www.spiral.net.

[14] M. Kumm, P. Zipf, M. Faust, and C.-H. Chang, “Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication,” in *Circuits and Systems ISCAS, IEEE International Symposium on*, May 2012, pp. 49–52.

[15] M. Kumm, “Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays,” *Ph.D. thesis, University of Kassel, Springer*, 2016.

[16] PAGSuite Project Website. (2016) http://www.uni-kassel.de/go/pagsuite.

[17] P. Tummeltshammer, J. C. Hoe, and M. Puschel, “Time-Multiplexed Multiple-Constant Multiplication,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 9, pp. 1551– 1563, Sept 2007.

[18] J. Chen and C. H. Chang, “High-Level Synthesis Algorithm for the Design of Reconfigurable Constant Multiplier,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1844–1856, Dec 2009.

[19] K. Mo¨ ller, M. Kumm, B. Barschtipan, and P. Zipf, “Dynamically Reconfigurable Constant Multiplication on FPGAs,” in *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014, pp. 159–169.

[20] K. Mo¨ ller, M. Kumm, M. Kleinlein, and P. Zipf, “Pipelined Reconfigurable Multiplication with Constants on FPGAs,” in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014, pp. 1–6.

[21] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf, and U. Meyer-Baese, “Multiple Constant Multiplication with Ternary Adders,” in *Field Programmable Logic and Applications (FPL), 23rd International Conference on*, Sept 2013, pp. 1–8.

[22] G. Baeckler, M. Langhammer, J. Schleicher, and R. Yuan, “Logic Cell Supporting Addition of Three Binary Words,” *US Patent No 7565388, Altera Coop.*, 2009.

[23] J. M. Simkins and B. D. Philofsky, “Structures and Methods for Implementing Ternary Adders/Subtractors in Programmable Logic Devices,” *US Patent No 7274211, Xilinx Inc.*, Mar. 2006.

[24] D. R. Reddy, “Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort. Department of Computer Science,” 1977.

[25] OpenCores. (2016) http://opencores.org/project,ternary adder.

[26] K. Chapman, “Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources,” *Application Note: Spartan-6, Virtex-6 Family, 7 Series FPGAs, Xilinx Inc.*, 2014.

[27] *Virtex-6 FPGA Configurable Logic Block User Guide UG364 (v1.2)*, Xilinx Inc., 2012.
[28] F. de Dinechin and B. Pasca, “Designing Custom Arithmetic Data Paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2012.
[29] Xilinx Inc., *LogiCORE IP Multiplier v11.2, DS255*, 2011.
[30] FIRsuite. (2016) http://www.firsuite.net.
[31] M. Kumm, K. Mo¨ ller, and P. Zipf, “Dynamically Reconfigurable FIR Filter Architectures with Fast Reconfiguration,” *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 8th International Workshop on*, pp. 1–8, 2013.
[32] A. Croisier, D. Esteban, M. Levilion, and V. Rizo, “Digital Filter for PCM Encoded Signals,” *U.S. Patent No. 3.777.130*, 1973.
[33] K. Chapman, “Constant Coefficient Multipliers for the XC4000E,” *Xilinx Application Note*, 1996.
[34] M. Kumm, P. Zipf, M. Faust, and C.-H. Chang, “Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication,” in *IEEE Int. Symposium on Circuits and Systems (ISCAS)*, 2012, pp. 49–52.
[35] M. Kumm, K. Mo¨ ller, and P. Zipf, “Reconfigurable FIR Filter Using Distributed Arithmetic on FPGAs,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 2058–2061.
[36] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, “Architectural Modifications to Enhance the Floating-Point Performance of FPGAs,” *Very Large Scale Integration (VLSI) Systems, IEEE Trans-actions on*, vol. 16, no. 2, pp. 177–187, 2008.
[37] T. H. Pham, S. A. Fahmy, and I. V. McLoughlin, “Low-Power Cor-relation for IEEE 802.16 OFDM Synchronization on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 8, pp. 1549–1553, Aug 2013.
[38] F. de Dinechin and B. Pasca, “Large Multipliers with Fewer DSP Blocks,” in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 250–255.