Pipelined Reconfigurable Multiplication with Constants on FPGAs

Konrad Möller, Martin Kumm, Marco Kleinlein, Peter Zipf
Digital Technology Group
University of Kassel, Germany
Email: {konrad.moeller, kumm, marco.kleinlein, zipf}@uni-kassel.de

***Abstract*—This paper presents a new algorithm to automatically create pipelined run-time reconfigurable constant multipliers. Reconfiguration between several constants is achieved by merging optimized pipelined adder graphs using multiplexers. The adder graphs perform the required multiplications by using additions, subtractions and bit-shifts only. They are generated by an existing heuristic called RPAG. The resulting reconfigurable pipelined single and multiple constant multipliers can be used for time-multiplexed multiplication reducing the required FPGA logic resources. In contrast to earlier approaches aiming at application-specific integrated circuits (ASICs) we introduce pipelining and take special care of the size of the added multiplexers to obtain FPGA-optimized solutions. We can show that the achieved pipelined run-time reconfigurable constant multipliers on average only need about 77% of the slices compared to the best solutions based on the merging of adder graphs published so far.**

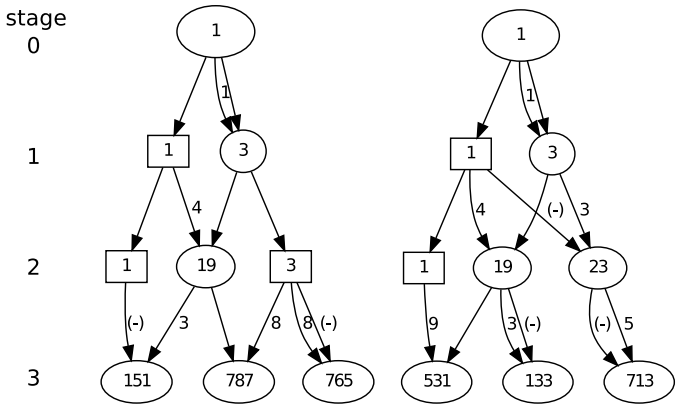
I. Introduction

This paper describes the automatic generation of run-time reconfigurable pipelined constant multipliers for field-programmable gate arrays (FPGAs). As constant coefficient multiplication is an essential operation in digital signal processing, finding optimized solutions for this operation is a well studied research topic. Although this paper focuses on reconfigurable multiplication on FPGAs, an overview on ASIC-related work is given first. On application-specific integrated circuits (ASICs) the so called single constant multiplication (SCM) and multiple constant multiplication (MCM) is done multiplier-less using additions, subtractions and bit shifts. This approach is also regularly applied to implementations on FPGAs (without any reconfiguration). Finding an optimal solution for SCM adder graphs is NP-complete [1]. Yet, optimal solutions could be found for constants of up to 12 bits [2], up to 19 bits [3] and up to 32 bits [4]. Beyond that there are good heuristics to generate SCM adder graphs (RAG-n, BHM [1] and H_{cub} [5]) for which generators and source code can be found online on the web page of the SPIRAL project [6]. A heuristic called RPAG which is specialized to generate pipelined adder graphs (PAGs) for SCM as well as for MCM was proposed in [7]. The source code of RPAG is also available online [8].

Run-time reconfiguration can be achieved by the insertion of multiplexers into SCM and MCM solutions. As this is a generalization of the basic SCM and MCM problem, the problem of finding a minimal reconfigurable SCM/MCM solution is NP-complete, too. Nevertheless, there are solutions to solve the problem of finding reconfigurable SCMs, called ReSCM in the following. An FPGA-specific algorithm is presented as ReMB method in [9] and was further analyzed and extended in [10]. An ReSCM is constructed from basic structures that fit into the basic logic elements (BLE) of FPGAs. Moreover there are three ASIC-optimized solutions. Tummeltshammer et al. [11] suggest to merge several optimized SCM graphs by a recursive algorithm called DAG fusion which fuses two SCM graphs with minimal hardware effort. Multiplexers are inserted to switch between the different constants. More than two coefficients can be included by recursively adding the related SCMs to the existing ReSCM in the same way. Chen et al. [12] exploit similarities between different coefficients using a canonical signed digit (CSD) representation to realize ReSCMs. The number of required adders is reduced by common subexpression elimination (CSE) through searching and fusing identical patterns in the CSD representation of constants. Multiplexers are inserted to switch between the different shifts and interconnections to realize a specific constant. Faust et al. [13] describe an algorithm which provides not only solutions for ReSCM but also for reconfigurable multiple constant multiplication (ReMCM). The authors also use an adder graph based approach with special focus on minimal logic depth.

II. Problem Statement and Contribution

Run-time reconfiguration of SCM and MCM circuits can be used to minimize resources by reuse of partial circuits as well as to construct adaptive filters (if reconfigurable constant multipliers are used as basic blocks). As we aim at pipelined realizations in this paper, we use the RPAG heuristic [7] as the starting point for our algorithm presented in Sec. III. Contrary to the other methods, RPAG is not forward-exploring reachable intermediate factors when searching for a step-wise constant composition but starts with the required output factors, trying to select those predecessors for the previous stage which reduce the adder depth and result in the lowest number of factors in that stage. The first approach to generate a reconfigurable pipelined single (RPSCM) and multiple constant multiplication (RPMCM) would thus be to construct it the same way: from the output stage to the input stage. This should be done with a set of possible topologies to determine the constants in the respectively preceding stage as this turned out to be very efficient in the RPAG heuristic.



(a) {151,787,765} (b) {531,133,713}

Fig. 1: RPAG solutions for the given constants.

This approach leads to a huge number of possible predecessors very quickly, which makes it inapplicable due to the resulting high complexity. Therefore another approach is needed.

The contributions of this paper are as follows:

- the idea of DAG fusion [11] is picked up to merge already optimized pipelined adder graphs (PAGs) generated with the RPAG [7] heuristic;
- in contrast to DAG fusion, all of the required PAGs are considered in one single optimization run to produce a better, multiplexer-aware pipelined realization;
- pipelined ReMCM solutions can be generated which is not possible with the original DAG fusion.

Our algorithm (Sec. III) provides implementations with very low resource usage (20% reduction) at a sometimes marginally reduced maximal clock frequency (Sec. IV).

III. Proposed Optimal Merging of Pipelined Adder Graphs

A. Starting Point

In order to realize the reconfigurable single and multiple constant multiplication the optimization starts with pipelined adder graphs generated with the RPAG heuristic. The optimization is explained by using an example with two configurations with the constant outputs $o_1 = (151, 787, 765)$ and $o_2 = (531, 133, 713)$. Their pipelined MCM realizations are shown as graph representation in Fig. 1. Each node in the graph corresponds to either an registered adder or subtractor (circular node), a register (rectangular node) or the input node (top node). Each edge is associated with a left shift and a sign. The value of each node corresponds to a factor, i. e., node 3 is obtained by shifting the input by one and adding the unshifted input: $3x = 2x+x$. The input of the proposed algorithm are the constants which have to be realized in the same output node. This depends on the desired application and is not part of the optimization. For the example we want so generate the output pairs $\binom{151}{531}$, $\binom{787}{133}$, $\binom{765}{713}$. Different from the example there can be more than two configurations. Examples with up to nine configurations for reconfigurable single constant multiplication are shown in Section IV.

B. Merging Principle

The first step is to determine which intermediate values in the preceding stage have to be calculated in the same adder to minimize the resulting overhead of possibly necessary multiplexers or switchable adder/subtractors (addsub). For a single stage with two configurations, this corresponds to the well known allocation problem for which exact solutions would be available. As we are interested in more than two configurations and several stages, we propose a novel method based on branch-and-bound. For this we have to evaluate all possible combinations. An example for this evaluation is shown in Fig. 2. It shows all possible combinations of the nodes in stage 2 of configuration o_1 with all possible nodes in stage 2 of configuration o_2. The nine results are partly mutually exclusive building six triplets of possible mappings. For example (a),(e) and (i) is one of these triplets. All triplets can be found in Fig. 4.

It can be seen in Fig. 2 that there are different numbers of multiplexers as well as different reasons why they are needed in the particular combination. For example in (a) the multiplexer is caused by a different shift, in (b) the multiplexers are caused by the fact that the inputs of the adders come from different predecessor nodes. From the local point of view these predecessors are unknown which is marked with a question mark in Fig. 2. Their contribution to the overall overhead will be considered in an other combination. For further details on that topic see the next section.

C. Cost Evaluation

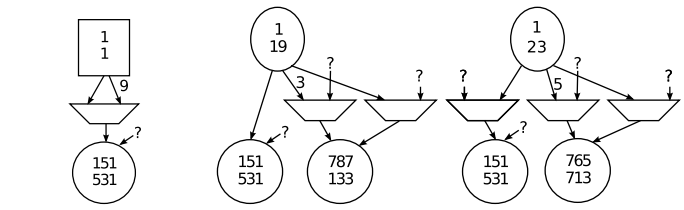
The cost of each of the nine combinations can be calculated separately. For the example each used multiplexer input has a cost value of $c = \frac{1}{2}$ which corresponds in this special case to half of a 2:1 multiplexer. Assuming that the multiplexers will be realized as a cascade of 2:1 multiplexer, for the general case each multiplexer input adds costs of

$$\text{cost}_{\text{MUX}} = \frac{\#\text{configurations} - 1}{\#\text{configurations}} \tag{1}$$

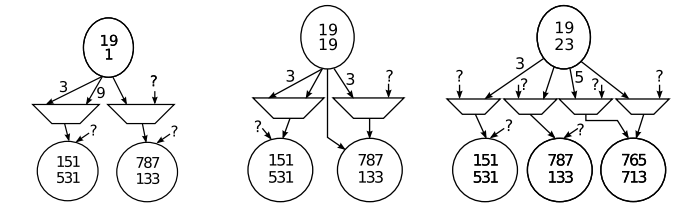
The cost for the combinations are stored in a matrix shown in TABLE I. It is two dimensional (resulting from 2 configurations) in the example but can be multidimensional in general. It assigns the nodes of configuration o_1 to the nodes of configuration o_2 with the corresponding cost value (left table). Additionally the relation to Fig. 2 (a)-(i) is shown (right table).

The costs of the unknown inputs (marked with '?' in Fig. 2) do not have to be considered when the cost for each combination is evaluated, because in the assembled triplet which builds a valid mapping there will be no unknown inputs. This can be easily seen in the mapping (a),(e),(i) in Fig. 3. The former unknown inputs complement each other resulting in four multiplexers. This corresponds to the sum of the cost values of (a),(e) and (i) in TABLE I.

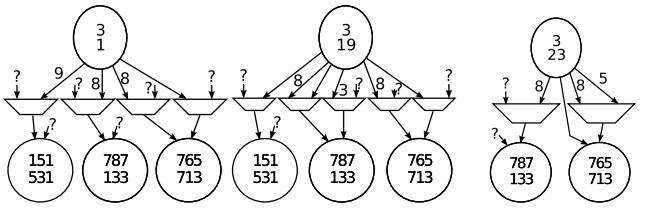
The knowledge about the partly mutual exclusion and the costs is summed up in a decision tree (Fig. 4). The overall cost evaluation and the search for the optimal solution is done by evaluating this decision tree in a depth-first manner.



(a) '1' o_1, '1' o_2 (b) '1' o_1, '19' o_2 (c) '1' o_1, '23' o_2



(d) '19' o_1, '1' o_2 (e) '19' o_1, '19' o_2 (f) '19' o_1, '23' o_2



(g) '3' o_1, '1' o_2 (h) '3' o_1, '19' o_2 (i) '3' o_1, '23' o_2

Fig. 2: Combination of the nodes in stage 2 of configuration o_1 with all possible nodes in stage 2 of configuration o_2.

TABLE I: Cost matrix (left) and relation to Fig. 2 (right) for stage 2 of the given example with nodes of configuration o_1 assigned to the nodes of configuration o_2.

	1	19	23
1	1	1	1.5
19	1.5	1.5	2
3	2	3	1.5

	1	19	23
1	(a)	(b)	(c)
19	(d)	(e)	(f)
3	(g)	(h)	(i)

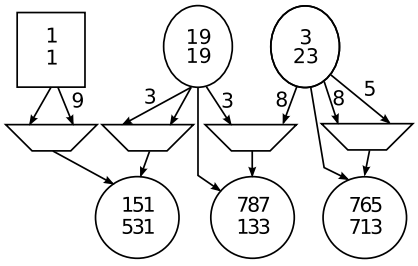


Fig. 3: Assempled mapping (a),(e),(i).

As an alternative to counting the number of used multiplexers also the word sizes in combination with the number of required mutliplexers, adders, subtractors, addsubs and registers can be used to evaluate the cost for each decision. In this case the unknown input matters as it directly affects the word size in the afterwards assembled multiplexer. That's why the word size has to be estimated in those cases, which can lead to an overestimation of the real costs.

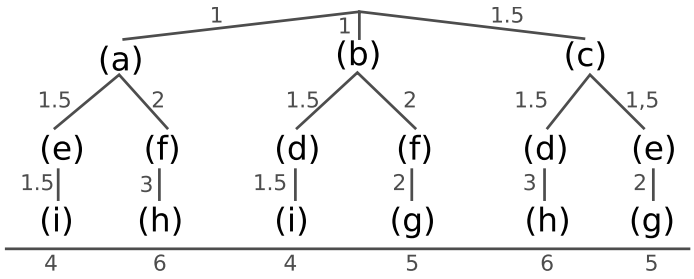


Fig. 4: Decision tree for stage 2 of the example. Each multiplexer input has cost of $\frac{1}{2}$.

D. Generating the Reconfigurable Pipelined Adder Graph

Each of the six triplets of Fig. 4 can be chosen to recursively determine the best combinations of the previous stages with the selected mapping. This adds an additional decision tree for each stage and each chosen mapping at the end of the stages' decision trees until the input stage is reached. As this can be done with all mappings it is possible to perform a complete search over all valid combinations by combining the decision trees of every stage to one single tree for the whole reconfigurable adder graph. This is very time and memory consuming as the number of combinations grows factorial with the number of adders in one stage multiplied by the number of combinations of the other stages for each combination. However, the presented approach exploits the following properties to find the optimal solution:

1) One does not have to know the whole decision tree but only the branch one is currently in. This moderates the memory problem, because only the local environment has to be stored.
2) There are many branches which are not relevant, because they are much too expensive. This is one of the key properties which is required for a branch-and-bound algorithm. Branches can be pruned whenever the currently best cost value is exceeded. So, firstly we do a greedy search by selecting the best result for each stage to get a first value for the cost of the whole reconfigurable adder graph. Then the algorithm starts to go through the whole decision tree branch by branch. Whenever the costs are higher than the current best cost the currently considered branch is pruned.

E. The Resulting Design

The best resulting reconfigurable multiple constant multiplier resulting from the proposed algorithm with the given configurations is shown in Fig. 5. We inserted pipeline registers after each stage which includes registers in the multiplexer stages. The pipeline registers in the adders and multiplexers are not shown in the figure. The values for the different

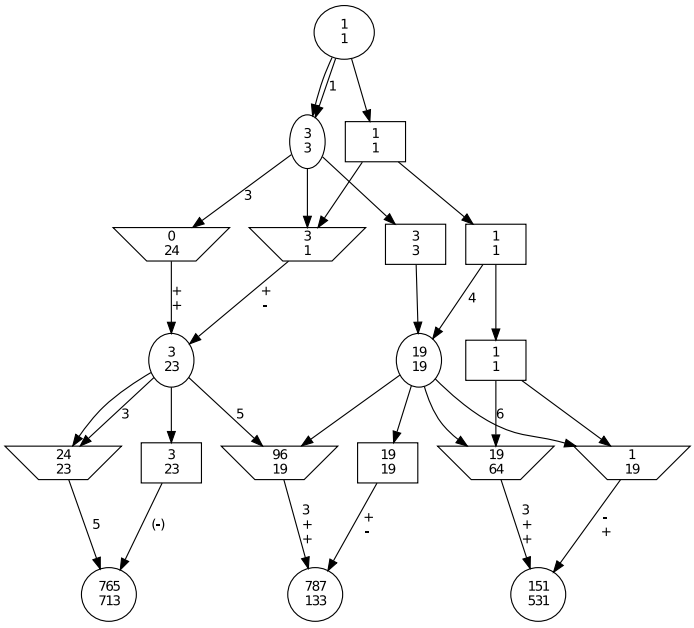


Fig. 5: Resulting RPMCM graph.

configurations are noted as vectors $\binom{o_1}{o_2}$. This is also valid for addsubs where the specific signs are depicted at the input arrows as vectors.

F. Additional Elements of the Algorithm

The final graph generated by the proposed algorithm consists of adders, subtractors, registers, multiplexers and addsubs. The addsubs result from the mapping of adders and subtractors of the original PAGs in one node. Another reason for addsubs is the mapping of subtractors only but with the negative input on different inputs of the original subtractors. This kind of addsub can be prevented by switching the inputs of the subtractors during optimization. Thus, it is possible to map all negative inputs to the same input resulting in a subtractor instead of a switchable adder-subtractor. Changing the inputs can on the one hand reduce the number of addsubs but on the other hand lead to an additional multiplexer. So a tradeoff has to be found. This is done by the alternative cost evaluation which was introduced in Sec. III-C, which considers addsubs, too.

The PAGs in Fig. 1 have the same number of nodes in each stage. This is, of course, not the case for all PAGs. For the stages in the PAGs with a lower number of nodes, virtual nodes are inserted for each missing node and marked as 'don't care'. That means that they have no cost as they will not lead to any multiplexer or addsub and do not have to be considered in the optimization process. This ensures that the optimization can be done without drawbacks with such PAGs.

Some multiplexers in the generated graph only switch between different shift values. There are cases in which these shift values are all greater than zero (c.f. Fig. 2 (i), right multiplexer). This leads to an unnecessary large word size in the affected multiplexer which should normally be detected by the synthesis. To ensure that the shifts get normalized we search for such cases after the optimization and normalize the shifts. That means we search for the smallest shift which is then subtracted from all input shift values. To get the correct final result the subtracted shift will be performed after the multiplexer. To illustrate this Fig. 6 contains a normalized version of Fig. 2 (i). The shift by 5 bits is performed after the multiplexer to reduce the multiplexers (and pipeline registers) word size by 5 bits. The word size reduction can also be seen in the last stage of Fig. 5.

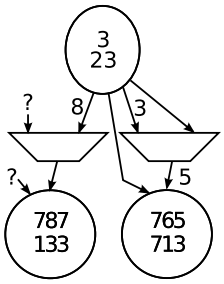


Fig. 6: Normalization of mutliplexer input shifts in Fig. 2 (i).

IV. Results

This section comprises an experimental evaluation of the proposed algorithm. First of all we compare the proposed approach to the DAG fusion algorithm [11]. In a second step the results are analyzed and differences in the resulting RPSCMs are shown.

A. Comparison to DAG Fusion for RPSCM

We compare the proposed algorithm to the DAG fusion algorithm as it also relies on the merging of adder graphs and is the the state-of-the-art method. Since no pipelining was used in the original algorithm we had to add it to get a fair comparison. So we took the DAG fusion source code which is available online [6], generated reconfigurable single constant multipliers and added pipeline registers after each adder, subtractor, addsub and multiplexer like it is done in the proposed algorithm. Moreover registers where inserted when the current inputs depended on a stage before the last stage to get a valid reconfigurable *pipelined* SCM graph according to an ALAP scheduling.

The set of input constants that we use was already taken for comparison in the earlier papers on run-time reconfiguration of constant multiplication ('A'-'D') in [11], [9], [12]. In addition to that we took the constants from Fig. 5 from [12] and call it set 'E'. The input sets 'F' and 'G' are the constants from Fig. 6 and 8 in [11].

The input of the proposed algorithm was created using the RPAG heuristic [8]. We created an MCM graph that realizes the particular input set for the RPSCM graph. For each constant we extracted the corresponding SCM adder graph. Thus, the adder minimization of RPAG is exploited. Then we automatically generated RPSCM graphs with the proposed algorithm.

We developed a VHDL code generator which is based on the FloPoCo library [14] to create synthesizable VHDL

TABLE II: Comparison of the proposed algorithm to DAG fusion (pipelined). Synthesis results after place and route. Best values are written in bold characters.

	proposed algotihm		DAG fusion (pipelined)	
Set	Sl.	f_{clk} [MHz]	Sl.	f_{clk} [MHz]
A	**38**	**626.57**	41	478.24
B	**32**	590.32	55	**593.12**
C	**24**	**606.43**	36	588.24
D	**23**	**599.52**	25	592.42
E	**46**	**575.37**	51	505.05
F	**18**	576.70	20	**628.14**
G	**11**	**631.71**	21	576.70
avg.:	**27.43**	**600.95**	35.57	565.99

code for both algorithms. For both methods (DAG fusion and the proposed algorithm) we mapped the designs to a Virtex 6 FPGA (xc6vlx75t-2ff484-2) using Xilinx ISE v13.4. The required slices and the maximum clock frequency after place and route are listed in TABLE II. The best values are written in bold characters.

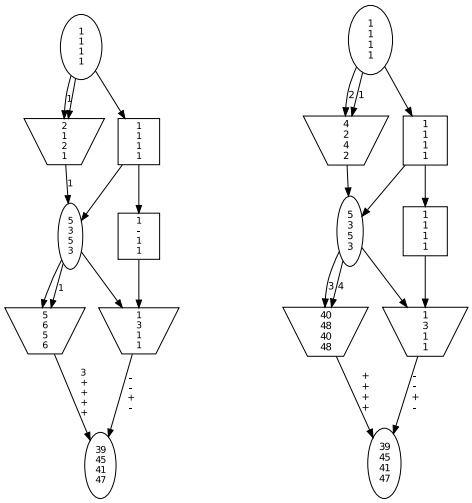
It can be observed that the proposed algorithm leads to the least number of required slices in all cases for the considered constant sets. Taken as a whole only about 77% of the number of slices is required with the proposed solution on average. The overall maximum clock frequency is 6% larger on average, but the best solutions in terms of the maximum clock frequency are spread over both algorithms. This results from the fact that the critical path in both solutions can be found in either one adder, one subtractor, one addsub or multiplexer with varying size due to pipelining.

B. Analyzing the Results

In this section the differences between the solutions will be analysed by the help of resulting RPSCM graphs for each algorithm. The first graphs are the results for input set D (see Fig. 7) which can be switched between four inputs. We selected them, because they need almost the same number of slices. The first interesting oberservation is that they look very similar although they were generated with two different algorithms. But there is a slight difference which reduces the number of slices. It is the normalization of the shifts explained in section III-F, which can be seen in the first and third stage.

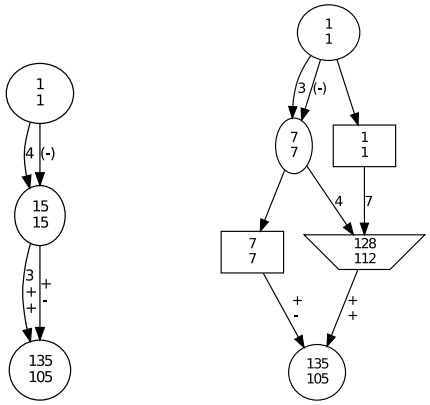
The second pair of solutions is the result of input set G (see Fig. 8). We selected this set, because it is a very small one but despite that the proposed solution only needs about half the number of slices. The proposed solution does not need any multiplexer, because it can be switched between the two output constants by changing the sign of the addsub.

The third example, input set B, leads to the largest number of slices for DAG fusion an needs 70% more slices than the proposed solution. One reason for this large difference is the different number of pipeline stages, resulting from adder depth optimization in the RPAG heuristic. A second reason is the larger number of multiplexers and registers which can be seen in Fig. 9 (b). The number of pipeline registers is no problem in the proposed solution Fig. 9 (a). In contrast to the RPAG



(a) proposed algorithm (b) DAG fusion

Fig. 7: Resulting RPSCM graphs for input set D.



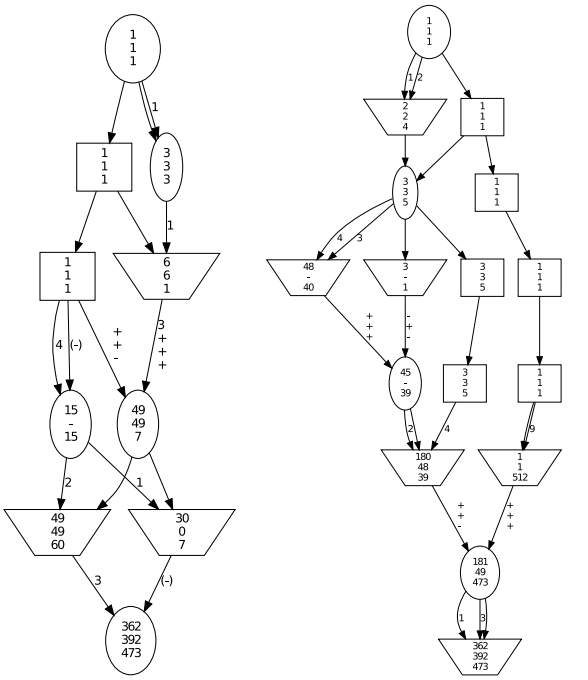
(a) proposed algorithm (b) DAG fusion

Fig. 8: Resulting RPSCM graphs for input set G.

heuristic, DAG fusion is not optimized for pipelining so this oberservation is not a big surprise. Nevertheless this example shows that is worthy to have an optimization of run-time reconfigurable pipelined constant multipliers which considers pipelining from the beginning.

V. Outlook

The results justify to put more effort into the FPGA-specific optimization of reconfigurable pipelined adder graphs. There are several ideas what should be added to the proposed algorithm in future work. The most interesting point is an adaption of the RPAG heuristic [7]. The RPAG heuristic minimizes the overall adder depth and the number of adders. This is not the best strategy for the generation of reconfigurable pipelined adder graphs as it is also important to consider the shifts and connections between the adders. Beyond that the different RPAG solutions, which are used for RPMCM are all generated independently without any knowledge of the other required pipelined adder graphs. For our purpose the solutions should not be independent to use similarities between the different solutions. This principle could be partly used when we created



(a) proposed algorithm (b) DAG fusion

Fig. 9: Resulting RPSCM graphs for input set B.

the RPSCM graphs for the experiments in Sec. IV, but is not applicable for RPMCM. So the idea is to take the original RPAG heuristic and to favour intermediate constants which can possibly also be used in other pipelined MCM graphs. Moreover the resulting shifts have to be considered during the optimization.

The RPAG heuristic delivers solutions that use adders with three inputs (ternary adders) [15] to reduce the number of operations and thus reduce the required hardware. Ternary adders are supported in recent Altera and Xilinx FPGAs [16], [17]. The usage of these three-input adders can also be integrated in the proposed algorithm by adapting the merging step to also reduce the number of operations and thereby multiplexer inputs.

The third extension will directly effect the required hardware. In Fig. 5 there is a multiplexer which switches between 24 and zero. A general observation is that it is a common case that a multiplexer input is zero. This results from the merging of a register with an adder. As all multiplexers are already followed by a register, it is planed to consider this special case in the VHDL code generation and to reset the register for the given configuration instead of switching to zero. This will help to further reduce the number of multiplexer inputs and thus reduce the required hardware.

VI. Conclusion

This paper presented an algorithm to generate pipelined runtime reconfigurable constant multipliers. It is based on the merging of pipeline optimized adder graphs generated with the RPAG heuristic. Multiplexers are introduced to switch between different constant multiplications in the merged reconfigurable adder graph. Besides the generation of reconfigurable pipelined single constant multipliers (RPSCM) the shown algorithm is also able to merge reconfigurable pipelined multiple constant multipliers (RPMCM). The results of the proposed algorithm were compared to pipelined results from the ASIC-optimized ReSCM algorithm DAG fusion by the help of synthesis results. These synthesis results were obtained by automatically generated VHDL implementations of the regarded RPSCMs. The results of the proposed algorithm only need 77% of the number of slices on average compared to the pipelined solutions of DAG fusion. The differences in the solutions were analyzed by means of some resulting RPSCM graphs. That way the advantages of the proposed FPGA-specific optimal algorithm could be pointed out.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] SPIRAL-Project. (2014) http://www.spiral.net.

[7] 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.

[8] RPAG Project Website. (2014) http://www.uni-kassel.de/go/rpag.

[9] 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.

[10] K. Möller, 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.