Advanced Compressor Tree Synthesis for FPGAs

Martin Kumm and Johannes Kappauf
University of Kassel, Digital Technology Group, Germany
Email: {kumm@uni-kassel.de, johannes.kappauf@student.uni-kassel.de}

Abstract—This work presents novel methods for the optimization of compressor trees for FPGAs as required in many arithmetic computations. As demonstrated in recent work, important key elements for the design of efficient but fast compressor trees are target-optimized 4:2 compressors as well as generalized parallel counters (GPCs). However, the optimization of a compressor tree for minimal resources using both compressors and GPCs has not been addressed so far. As this combined optimization is a non-trivial task, three methods are proposed to find best solutions for a given problem size: 1) a heuristic that obtains compressor trees with typically less resources and fewer stages than state-of-the-art heuristics, 2) an integer linear programming (ILP)-based methodology that finds optimal compressor trees using the fewest stages possible, 3) a combined approach that partially solves the problem heuristically to reduce the search space for the ILP-based method. In all methods, the cost for pipeline registers can be included. Synthesis experiments show that the proposed methods provide pipelined compressor trees with about 40% less LUTs compared to trees of 2-input adders at the cost of being about 12 . . . 20% slower.

1 Introduction

The addition of several variables is used in nearly any arithmetic operation. Compressor trees deliver a fast and compact realization to common adder trees by using carry-save arithmetic. The most prominent example is the multiplication, where several partial products have to be added. However, there are much more applications besides multiplication. In a network of multiply and add operations many operations can be merged into single compressor trees following the well known concept of merged arithmetic [1], [2]. This includes basic operations like squaring, multiply-add and complex multiplications but also function evaluation by using polynomials as well as signal processing applications like digital filters and linear transforms, just to name a few. In addition, several methods have been proposed which try to maximize the use of carry-save arithmetic in general computations [3]–[5].

The use of compressor trees has a long history in the design of arithmetic units for microcomputers [6], [7]. The basic idea is to avoid the slow carry propagation by passing (saving) the carry to the next compressor stage instead of propagating it within the same stage. While this guarantees substantial speed-ups in application specific integrated circuits (ASICs) or custom ICs, the use of carry save arithmetic on field programmable gate arrays (FPGAs) was regarded as unsuitable for a long time. The reason for this are the dedicated carry chains found in modern FPGAs which are significantly faster than a generic wire that is mapped to the FPGA's routing fabric. However, it was first shown by Parandeh-Afshar et al. [8], [9], that a significant delay improvement can be obtained by using compressor trees on FPGAs. The key was to adapt the concept of so-called generalized parallel counters (GPCs) [10] (for further details, see Section 2.1) to FPGAs, which provide a better utilization of the look-up tables (LUTs). They achieved delay reductions of about 30% while having a slight resource overhead of 5%. However, the design of the compressor tree is much more complex compared to the simple classic algorithms from Dadda [7] or Bickerstaff [11]. They provided a heuristic [8] as well as an exact integer linear programming (ILP) method [9]. Later, the same group provided a simplified heuristic and proposed to use GPCs which include the fast carry chain which provides an even better utilization [12], [13]. They achieved delay reductions of 33% (Xilinx Virtex 5) and 45% (Altera Stratix III) and a similar resource usage compared to state-of-the-art adder trees built from ternary adders. Since then, several advanced GPCs have been proposed that effectively use the FPGA carry-chain [14]–[16].

A completely different approach is the design of regular compressor trees using row compressors like the 4:2 compressor [14], [17], [18]. Unlike counters or GPCs which reduce one or more columns of same weight, 4:2 compressors take four rows and compute two rows in a redundant representation. A 4:2 compressor that was optimized for Xilinx' four-input LUT FPGAs with fast carry chains (Virtex 2/4 and Spartan 2/3) was proposed by Ortiz et al. [17]. They could utilize the fast carry chain to map two cascaded full adders into a single slice which forms the basic element of a 4:2 compressor. A similar approach considering different redundant number systems was developed by Kamp et al. [18]. A more efficient mapping of 4:2 compressors to the modern 6-input FPGAs of Xilinx was proposed by us in [14]. Here, the two full adders (FAs) are mapped to the same

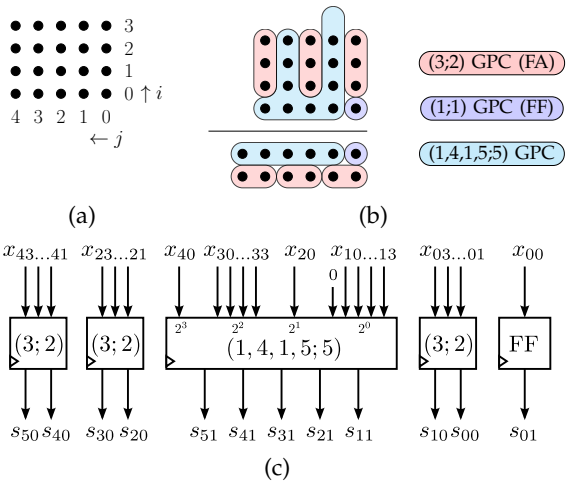


Fig. 1: Example dot diagrams: (a) 4-input addition of 5-bit variables; (c) compressor tree solution in the dot diagram; (c) resulting compressor tree

LUT like done in a ternary adder leading to half of the resource usage compared to [17], [18].

Another idea of utilizing the fast carry-chain to build not just basic compressors as in [17], [18] but complete compressor trees was proposed by Hormigo et al. [19]. They use a linear array structure of carry-save adders (CSAs) from which preceding stages of FAs can be mapped to conventional ripple carry adders (RCAs) or ternary adders. As these carry-chains are much shorter compared to a conventional adder tree built from RCAs, the delay is much shorter while requiring similar resources. It appears to be the fastest choice for combinatorial multi-input adders but may be less efficient for irregular shaped compressor trees where unused inputs have to be set to zero.

An ILP-based compressor tree optimization targeting the power and delay optimization of compressor trees was proposed recently by Matsunaga et al. [20], [21]. They could drastically reduce the number of variables and constraints in the ILP formulation leading to a reduced optimization time. A similar approach was developed independently in a recent work of our group [15], where the focus was on pipelined compressor trees. The motivation for optimizing pipelined compressor trees comes from the following two facts: First, the large routing delays of FPGAs require the introduction of pipeline registers to achieve the desired throughput and, second, the cost of flip flops (FFs) have to be considered in the optimization. Indeed, the effective FF costs strongly depend on the usage of GPCs as the pipeline registers in GPCs typically come for free (the corresponding FFs in that logic unit are unused otherwise) and only FFs to balance the pipeline have to be considered.

The compressor tree optimization of [15] was integrated in the 'bit heap' framework [22] of the FloPoCo arithmetic core generator [23]. This framework provides classes to define, optimize and generate compressor trees which are internally represented like dot diagrams as a heap of bits, providing a great abstraction from a software engineering point of view. Many arithmetic operators provided in FloPoCo use this framework. However, it became quickly clear that many of these cores have problem sizes which become intractable for the exact ILP-based methods [15], [20]. Due to the fact that it was shown that there is a substantial gap between the optimal solution and heuristic approaches [15], [20], [21], one goal of this work is to provide methods which can handle large problem sizes while still providing a high quality. As the 4:2 compressor is one of the most efficient compressor [14], another goal of this work is to provide a methodology to incorporate this class of compressor in the optimization. Our contributions to achieve this are as follows:

1) An improved heuristic for optimizing compressor trees using GPC and row adders like the 4:2 compressor that results in less resources and fewer stages on average compared to a previous heuristic [8].
2) An extended ILP formulation for minimum stage count that significantly speeds up the optimization.
3) Another ILP extension that allows the inclusion of row adders.
4) A combined approach that uses the heuristic for partial optimization for problem reduction while leaving the critical parts for the ILP solver.

2 Background

The synthesis of compressor trees is best explained by introducing the dot-representation of the problem. As a running example, an unsigned multiple-input addition is used which is given as

$$S = \sum_i X_i = \sum_i \sum_j 2^j x_{i,j} \ . \tag{1}$$

Each X_i represents a bit vector with $x_{i,j}$ denoting a single bit of weight 2^j. The compressor tree problem can be represented in a dot-diagram, where each bit which has to be added is represented as a single dot. An example of the addition of four variables with five bit each is shown in Fig. 1(a). By convention, the rightmost dot represents weight 2^0 with increasing weight to the left. The order of the bits within a column does not matter. During the synthesis of a compressor tree dots are removed in the dot diagram by applying compressors like GPCs which are introduced in the following.

2.1 Generalized Parallel Counters

A *counter* is a circuit that counts the number of input bits which are one. They are typically denoted as $(p; q)$ or $p : q$ counter, where p is the number of input bits having all the same weight while q is the number of output bits which represent the number of inputs which are one in binary. From this definition, the number of

TABLE 1: High efficient GPCs and adders targeting Xilinx FPGAs (where τ_{L}, τ_{CC} and τ_{R} describe the delay of a LUT, one bit of carry propagation in the carry chain and a local routing delay, respectively, with $\tau \approx \tau_{\mathrm{L}} \approx \tau_{\mathrm{R}}$)

GPC/ row adder	Ref.	#LUT6 (k)	Efficiency ($E = \delta/k$)	Delay
(6;3)	[22]	3	1	$\tau_{\mathrm{L}} \approx \tau$
(1,5;3)	[22]	3	1	$\tau_{\mathrm{L}} \approx \tau$
(5;3)	[25]	2	1	$\tau_{\mathrm{L}} \approx \tau$
(1,4;3)	[25]	2	1	$\tau_{\mathrm{L}} \approx \tau$
(2,3;3)	[14]	2	1	$\tau_{\mathrm{L}} + 2\tau_{\mathrm{CC}} \approx \tau$
(3;2)	[22]	1	1	$\tau_{\mathrm{L}} \approx \tau$
(1,4,1,5;5)	[14]	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(1,4,0,6;5)	[14]	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(1,3,2,5;5)	[15]	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(6,2,3;5)	[16]	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(6,0,6;5)	[15]	4	1.75	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(6,1,5;5)	[16]	4	1.75	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
2-input add.		k	1	$\tau_{\mathrm{L}} + k\tau_{\mathrm{CC}}$
ternary add.		k	$2 - \frac{2}{k}$	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + k\tau_{\mathrm{CC}} \approx 3\tau + k\tau_{\mathrm{CC}}$
4:2 comp. (a/b)	[14]	k	$2 - \frac{2}{k}$	$\tau_{\mathrm{L}} + k\tau_{\mathrm{CC}}$

output bits can be at most $q = \lceil \log_2(p+1) \rceil$. A full-adder is an example of a $(3;2)$ counter as it has three inputs (the two arguments and the carry-in) and two output bits (sum and carry out). GPCs (which are also called multicolumn counters [24]) allow that input bits may have different weights (and are thus located in different columns). A GPC is commonly denoted as tuple $(p_{n-1}, p_{n-2}, \ldots, p_0; q)$, where p_j represents the number of input bits of weight 2^j and q is the number of output bits. A (3,5;4) GPC, for example, computes the sum of three input bits of weight two plus five input bits of weight one. The result is a number in the range $0 \ldots 2 \cdot 3 + 5 = 11$ and is represented by a single bit vector with 4 bit.

An important metric to evaluate GPCs is their *efficiency* [22] (also called *area degree* in [13]) which is defined as the quotient of the number of removed bits and the number of required LUTs k

$$E = \frac{b_i - b_o}{k} = \frac{\delta}{k} \tag{2}$$

where b_i and b_o denote the number of input and output bits, respectively. High efficient GPCs typically have the property that their input count may be highly irregular, like the $(6, 0, 6; 5)$ GPC [15].

Another GPC metric is the *compression ratio* [8] (also called *compression factor* [22]), which is defined as the ratio of input to output bits

$$\gamma = \frac{b_i}{b_o} \ . \tag{3}$$

Previous work indicated that a higher compression ratio leads to fewer stages in the compressor tree [8], [22].

A list of GPCs used in this work is given in the upper part of Table 1. Their corresponding dot transformations (i. e., the dots they remove/produce) are visualized in Fig. 2.

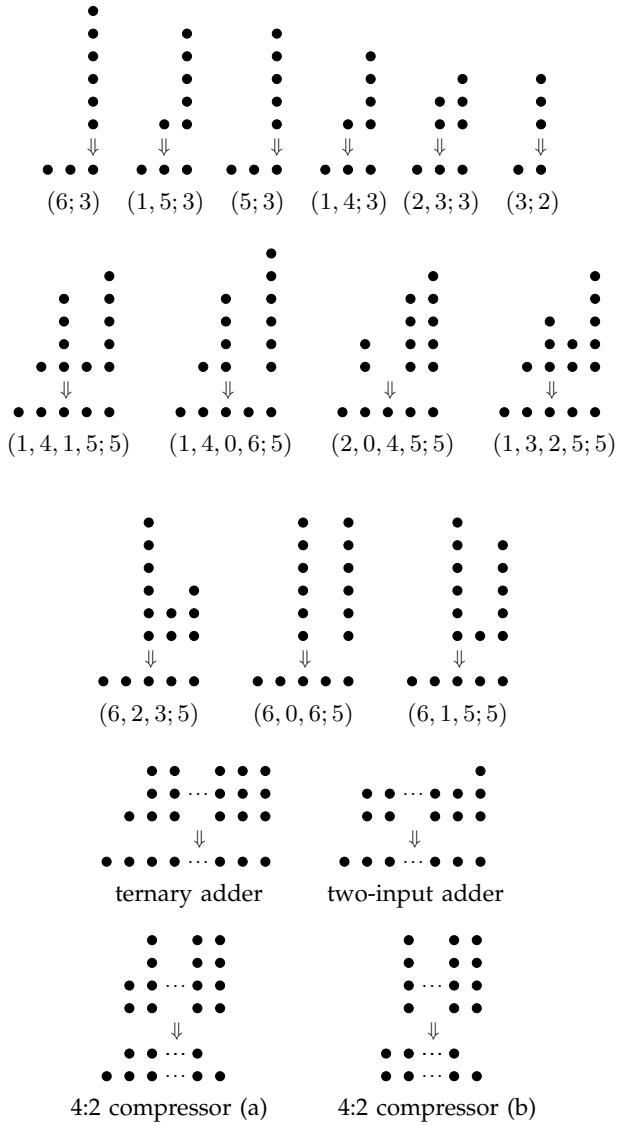


Fig. 2: Dot transformations for the GPCs and row adders from Table 1

2.2 Row Adders

Unlike GPCs which compress a fixed number of columns, common two-input adders, ternary adders or the 4:2 compressor discussed above perform a reduction by rows and, hence, have a variable number of columns [24]. Although adders can be constructed in compressor trees by primitive compressors like full adders, it is beneficial for the optimization to treat them with their specific cost separately. In the following, we refer to these just as *row adders*.

The adders for row compression considered here are characterized in the lower part of Table 1 and also visualized in Fig. 2. It should be noted that while the common two-input adder has a rather low efficiency, the ternary adder as well as the two possible variants of the 4:2 compressor (a and b) [14] have the best efficiency at all which reaches $E = 2$ for output word size $k \to \infty$. However, their shape may be unsuitable

for irregular (non-rectangular) dot diagrams. Also note that the ternary adder is the slowest circuit among the considered adders. This comes from the fact that, compared to a two-input adder, an additional LUT as well as a local routing for intermediate carries is included in the critical path. A detailed evaluation of ternary adders on different FPGAs is given in [26].

2.3 Compressor Tree Synthesis

Each GPC or row adder can now be regarded as a transformation in the dot diagram. Applying a $(1, 5; 3)$ GPC in the least significant column of Fig. 1(a) would remove 5 bits in column 0, one bit in column 1 and would produce three new bits in a row. When all bits are *covered* by GPCs, this stage of compression is complete and the same can be performed with the produced output bits. The procedure ends when only two (three) rows remain, which are compressed using a standard two-input (ternary) carry propagate adder (CPA). Fig. 1(b) shows the optimal solution (i. e., least LUTs) using the GPCs of Table 1 for the example in Fig. 1(a). Each bit is covered by a GPC while one GPC input is not connected (set to '0'). From that mapping, the corresponding compressor tree can be directly created as shown in Fig. 1(c). In this example, s_{ij} denotes the resulting bit i in column j. The final result is obtained by adding the two remaining rows by using a common two-input adder.

There is a multitude of ways in realizing a compressor tree, most of them using different resource cost, delay and latency. The corresponding optimization problem which is considered in this work is to find a compressor tree with the least resource cost that fulfills some additional constraints on delay or latency.

3 Previous Work

3.1 Heuristic Approach of Parandeh-Afshar et al.

A powerful heuristic for the design of compressor trees using GPCs was proposed by Parandeh-Afshar et al. [8]. This heuristic was later simplified by the same group [13]. Unfortunately, due to this simplification the heuristic can not be used together with the GPC set of Table 1 as it is assumed that the column size is decreasing with increasing weight. For example, a $(1, 3, 2, 5; 5)$ GPC is not allowed as $3 > 2$. Hence, we chose [8] as our baseline which is more general but is able to produce the same results using the same GPCs and metric. Algorithm 1 shows a pseudo code of the algorithm.

In the initialization phase (lines 3–8), an ordered list of GPCs is created based on the target FPGA. First, all so-called *covering* GPCs are created which can not be implemented by another GPC. From these, all possible *primitive* GPCs are derived for the given input/output word sizes. For example, a (1,5;3) GPC covers the primitive (1,4;3) GPC but not vice versa. The GPCs are then ordered by their compression ratio (Line 5). Note that in principle the other metrics defined in [13] can be used to order the GPC list.

Algorithm 1: Compressor tree heuristic from [8]

```
1  heuristicPA(inputColumns, targetFPGA, I):
2
3  gpcList = getCoveringGPCs(targetFPGA)
4  gpcList = findPrimitiveGPCs(gpcList,M,N)
5  gpcList = orderGPCsByCompRatio(gpcList)
6
7  s = 0
8  cols[0] = inputColumns
9
10 while max(cols[s]) > I do
11   do
12     c = maxHeightColumn(cols[s])
13     foreach gpc in gpcList do
14       if GPCfitsInCol(gpc,c) break
15     end
16
17     cols[s] = remDots(cols[s], c, gpc)
18     cols[s+1] = genDots(cols[s], c, gpc)
19
20   until all dots are covered or
21   no reasonable GPC can be found
22
23   cols[s+1] += cols[s]
24
25   s ← s+1
26 end
27
28 generateFinalCPA(cols[s-1])
```

The optimization strategy is then to select the column with the most number of bits (Line 12) starting with stage $s = 0$. Next, the first GPC from the ordered list that fits to this and the neighboring columns is selected (lines 13–15) and the corresponding dots from the current stage are removed (Line 17) and the produced output dots of the succeeding stage are generated (Line 18). The inner loop ends when all bits are removed or no further GPC can be applied. Then, this stage is complete, possibly remaining bits are added to the next stage (Line 23). The stage index s is incremented and the next stage is computed. The algorithm terminates when the height of each column is less than the number of inputs I of the CPA. This parameter depends on whether the FPGA supports ternary adders ($I = 3$) or only two-input adders ($I = 2$). At last, the final CPA is generated.

3.2 Previous ILP Formulation

A compressor tree optimization based on integer linear programming (ILP) was proposed in [15]. It can be solved by any standard ILP solver like the commercial Gurobi [27] or the open-source tool SCIP [28].

All used variables and constants are summarized in Table 2. The main idea is to count the number of bits (dots) in each column c for each stage s using variables $N_{s,c}$. The input problem is defined by setting the number of bits in stage zero ($N_{0,c}$) accordingly. Next, a set of integer variables $k_{s,e,c}$ denotes how many GPCs of type $e = 0 \dots E-1$ are applied in stage $s = 0 \dots S-1$ and column $c = 0 \dots C-1$. As the maximum stage count S is unknown, it has to be set to an upper bound [15]. Each

TABLE 2: Variables (top) and constants (bottom) used in ILP formulation

Variable/ Constant	meaning
$k_{s,e,c} \in \mathbb{N}_0$	Number of compressor e in stage s and column c
$c_e \in \mathbb{R}$	Cost (in LUTs) of compressor e
$N_{s,c} \in \mathbb{N}_0$	Number of bits in stage s and column c
$D_s \in \{0,1\}$	Decision variable that is '1' if and only if stage s is the final stage
$M_{e,c} \in \mathbb{N}_0$	Number of bits removed from compressor e in column c
$K_{e,c} \in \mathbb{N}_0$	Number of bits generated from compressor e in column c
$E \in \mathbb{N}_0$	Number of compressing elements
$C \in \mathbb{N}_0$	Maximum number of columns
$C_e \in \mathbb{N}_0$	Maximum number of columns of compressor e
$S \in \mathbb{N}_0$	Maximum number of stages

compressor e is characterized by the number of input bits $M_{e,c}$ that are removed and the number of output bits $K_{e,c}$ that are generated in column c, respectively. For example, if compressor $e = 4$ is a $(1,5;3)$ GPC, then $M_{4,0} = 5$, $M_{4,1} = 1$ and $K_{4,0\ldots2} = 1$. With this notation, also compressors with more than one output bit per column can be represented.

Using these variables, the following ILP formulation is used to optimally solve the compressor tree optimization problem:

$$\text{minimize} \sum_{s=0}^{S-1} \sum_{c=0}^{C-1} \sum_{e=0}^{E-1} c_e k_{s,e,c}$$

subject to

$$\text{C1:} \quad N_{s-1,c} \leq \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} M_{e,c+c'}\, k_{s-1,e,c+c'} + ID_s \quad \text{for } s = 1 \ldots S-1,\ c = 0 \ldots C-1$$

$$\text{C2:} \quad N_{s,c} = \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} K_{e,c+c'}\, k_{s-1,e,c+c'} \quad \text{for } s = 1 \ldots S-1,\ c = 0 \ldots C-1$$

$$\text{C3:} \quad N_{s,c} \leq \begin{cases} 2 + (1-D_s)I & \text{for two-input VMA} \\ 3 + (1-D_s)I & \text{for ternary VMA} \end{cases}$$

$$\text{C4:} \quad \sum_{s=1}^{S-1} D_s = 1$$

The objective is to minimize the resource cost. For that, the used GPCs are weighted by their corresponding LUT cost c_e. Now, the first constraint (C1) ensures that all bits in each column and stage except the output stage are connected to inputs of compressors. To exclude the output stage, the term ID_s is added where I is a large positive number such that $I > N_{s,c}$. In case $D_s = 1$, the constraint is always fulfilled which virtually disables this constraint (this is often called a *big-M* constraint). Constraint C2 simply computes the number of bits produced by compressors which are taken as input to the next stage. The column height of the output stage is constrained by C3. This constraint is disabled for all non-output stages, i. e., when $D_s = 0$. Constraint C4 ensures that there is exactly one stage that is marked as the output stage.

One key element in the model is that a single-input, single-output (1;1) *pseudo* GPC is included in the set of compressors. In a combinatorial compressor tree, it represents a simple wire. In a pipelined compressor tree, it represents a flip-flop for each pipelined stage which contributes real cost to the objective.

4 Proposed Methodology

The proposed method consists of three main contributions: First, an improved heuristic is suggested that is also capable in utilizing row adders. Next, an extension of our previous ILP formulation [15] is given that targets compressor trees with minimum stage count including row adders. Limiting the stage count turned out to also speed up the optimization or alternatively finds better solutions within the same time. Finally, the heuristic as well as the ILP approach are combined to yield better heuristic solutions which are close to the optimum.

4.1 Improved Heuristic

We observed a few limitations in the heuristic of [8]. One limitation comes from the strategy of the column selection: By always selecting the highest column, it sometimes occurs that only a compressor with poor efficiency fits to that column. This can later reduce the efficiency of other GPCs applied to that column. Furthermore, sorting the GPCs by their efficiency instead of their compression ratio turned out to be more effective without sacrificing the depth in most of the cases. Another practical observation was that computing all primitive GPCs results in a quite large number of GPCs when considering the high efficient GPCs as given in Fig. 2. For example, the $(6,0,6;5)$ GPC covers $7 \times 7 - 2 = 47$ different primitive GPCs, leading to hundreds of GPCs in the resulting GPC list. Another limitation is the lack of support for adders for row reduction. Again, all primitive GPCs for each possible column count could be enumerated, leading to an unacceptable large number of GPCs.

Our proposed heuristic circumvents these limitations. Its pseudo code is given in Algorithm 2. First, our heuristic considers only covering compressors (which can be GPCs or row adders) which are ordered by their efficiency (lines 3–4). As the widths of the row adders are not fixed, the maximum efficiency (see Table 1 for $k \to \infty$) is assumed for the ordering. Next, for each stage, the columns are ordered by their height. In case of equal height the column with the lower weight is preferred (Line 10). Now, the compressors are examined with decreasing efficiency. For each compressor and column, the resulting *effective* efficiency is computed (Line 16). To compute the *effective* efficiency, only the actually connected inputs b_i are counted in (2).

For row adders, the width is increased from $k = 2$ to the maximum number of columns. The efficiency drops at latest when the width is wider than that of the dot diagram. The highest efficiency of all the evaluated widths for this compressor is returned by `evalEffectiveEfficiency()`. The compressor with best efficiency is saved in lines 17–21. In case that the selected compressor is a GPC with an efficiency identical to the achieved *effective* efficiency, i. e., all inputs are used, the loop can be terminated (Line 23) as it is known that no better GPC will occur in the sorted list. For the same reason, we can terminate the next outer loop when the efficiency of the next compressor is lower or equal than the best efficiency found so far (Line 14). Once a compressor is found, the corresponding dots are removed and dots of the succeeding stage are generated (lines 29–30) like done in Algorithm 1. Line 27 provides a possibility to skip compressors which have an efficiency less than a minimum effective efficiency which can be specified for each stage s. This feature is used in the combined approach which is introduced in Section 4.3. For the heuristic, it is set to $\overrightarrow{\text{eff}}_{\text{min}} = (0, \ldots, 0)$. In contrast to Algorithm 1, we always include a *pseudo* $(1;1)$ GPC in the set of compressors. It is sorted to the end of the GPC list as its efficiency is defined to be zero. With that extension we are able to design efficient pipelined compressor trees as even a GPC with efficiency one (e.g., a full adder) is preferred to a flip-flop which has a similar cost but efficiency zero.

The worst case computational complexity is identical to the previous heuristic as the inner loop processing the columns (lines 15 to 24) has also to be performed in the `GPCfitsInCol` method of Algorithm 1 (Line 14). Both heuristics typically find a solution within a few seconds.

Algorithm 2: Proposed compressor tree heuristic

```
1  heuristicImp(inputColumns,targetFPGA,I,eff⃗min):
2
3  compList = getCoveringCompressors(targetFPGA)
4  compList = orderCompByEfficiency(compList)
5  s = 0
6  cols[0] = inputColumns
7
8  while max(cols[s]) > I do
9    do
10     cols[s] = orderColumnsByHeight(cols[s])
11     eff_best = 0
12
13     foreach comp in compList do
14       if(getEfficiency(comp) ≤ eff_best) break
15       foreach c in cols[s] do
16         eff = evalEffectiveEfficiency(comp,c)
17         if eff > eff_best do
18           eff_best = eff
19           comp_best = comp
20           c_best = c
21         end
22         if(type(comp) ≡ GPC and
23            getEfficiency(comp) ≡ eff_best) break
24       end
25     end
26
27     if(eff < eff⃗min(s)) break
28
29     cols[s] = remDots(cols[s], c_best, comp_best)
30     cols[s+1] = genDots(cols[s], c_best, comp_best)
31   until all dots are covered
32
33   s ← s+1
34 end
35
36 generateFinalCPA(cols[s−1])
```

4.2 Improved ILP Formulation

The ILP formulation in Section 3.2 has the problem that the maximum number of stages is unknown. In [15] it was overestimated by the stage count of a compressor tree consisting of full adders only which can be obtained from the Dadda sequence [7]. However, it is a rather rough estimate that can lead to many decisions that have to be evaluated during optimization. Experiments with a fixed stage count revealed that if the minimal stage count is known in advance, the optimization run-time can be drastically reduced. On the other hand, if an underestimation of the stage count is used, the solver is typically fast in proving its infeasibility (typically within a few seconds).

Hence, we propose to run several ILP optimizations, each with a fixed stage count starting from $S = 1$ which is increased when no feasible solution is found. With this simple methodology we can speed up the overall optimization time or can improve the quality that is obtained in a fixed given time. For that, the constraint C4 in Section 3.2 has to be removed and C1 & C3 have to be replaced with the following constraints:

$$\text{C1}': \quad N_{s-1,c} \leq \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} M_{e,c+c'}\, k_{s-1,e,c+c'} \quad \text{for } s = 1 \ldots S-2,\ c = 0 \ldots C-1$$

$$\text{C3}': \quad N_{S,c} \leq \begin{cases} 2 & \text{for two-input VMA} \\ 3 & \text{for ternary VMA} \end{cases}$$

Note that S now corresponds to the variable stage count and not to the upper bound.

4.2.1 Support for Row Adders

The ILP formulation above can not handle row adders except when tabulating all different adder widths as GPCs, which would lead to an excessive increase in variables and constraints. This can be avoided by splitting a compressor with variable size in several dependent compressors and adding constraints on how these parts are related. This idea was mentioned in [15] and is simplified to a single constraint in the following. For that, a row adder is divided into three partial compressors: e_L (placed at the column with lowest significance), e_M (placed at multiple columns in the middle) and e_H (placed at the column with highest significance).

As an example, take the 4:2 compressor (a) of Fig. 2. The lowest (rightmost) column compressor reduces four bit of the first column into one bit, corresponding to a compressor with $M_{e_L,0} = 4$ and $K_{e_L,0} = 1$. The middle compressors have four inputs and two outputs in each column ($M_{e_M,0} = 4$ and $K_{e_M,0} = 2$). The highest (leftmost) compressor has two input bits and produces two output bits in the same column and another bit in the next higher column ($M_{e_H,0} = 2$, $K_{e_H,0} = 2$ and $K_{e_H,1} = 1$). These partial compressors only work if they are connected in the right order (due to the internal carry propagation) which can be obtained by introducing the constraint

$$\text{C5:} \quad k_{s,e_H,c+1} + k_{s,e_M,c+1} = k_{s,e_M,c} + k_{s,e_L,c} \quad \text{for } s = 1 \dots S-1,\ c = 0 \dots C-1$$

(the $k_{s,e,c}$ is defined to be zero for columns out of range, i. e., for $c \neq 0 \dots C-1$). This constraint ensures that each compressor of type e_L or e_M in column c has an e_M or e_H compressor to its left (column $c+1$). By doing so, it is guaranteed that the correct shape of the row adder is obtained.

4.3 Combined Heuristic with ILP

In the previous two sections we proposed a heuristic and an optimal method based on ILP. As demonstrated in the results, the optimal method can now be applied to problems with a several hundreds of bits within two hours of computation time. For larger problems, the ILP solver is able to deliver heuristic solutions. In this section, we propose a combined optimization using the improved heuristic of Section 4.1 together with the ILP-based method of the previous section to further improve the quality of large size problems.

For that, we propose to pre-solve parts of the problem using the heuristic of Section 4.1 and to optimize the remaining problem which is reduced in size using the optimal ILP of Section 4.2. The resulting flow is depicted in Fig. 3. The heuristic passes the reduced problem as well as the partial solution to the optimal compression which computes the final solution. In fact, it turned out that the first stages have the least influence to the optimization quality. Of course, even if optimal parts are involved, the solution is a heuristic solution but with better quality for large problems within the same optimization time.

The heuristic pre-solving involves the following two strategies:

1) Solving complete stages by the heuristic.
2) Solving stages partially to reduce their size. To do so, we follow the strategy that the heuristic should apply compressors only until the effective efficiency is above a certain limit (eff_{min} in Algorithm 2).

Both strategies can be combined in a single parameter vector $\overrightarrow{\text{eff}}_{\text{min}}$ which specifies the minimum efficiency for each stage. Each element can be set one of the following three cases:

1) $\text{eff}_{\text{min}} = 0$: The corresponding stage is completely solved by the heuristic.
2) $\text{eff}_{\text{min}} > 0$: The heuristic stops when no compressor is found that has a minimum efficiency of at least eff_{min}. Hence, the corresponding stage is partially solved and the remaining problem is solved optimally using ILP.
3) $\text{eff}_{\text{min}} = \infty$: The corresponding stage is completely solved by the ILP method.

For example, the vector $\overrightarrow{\text{eff}}_{\text{min}} = (0, 1.5, \infty)$ would specify that the first stage ($s = 0$) is solved by the heuristic. In stage $s = 1$ only compressors with a minimum effective efficiency of 1.5 are applied and the remaining bits in stage $s = 1$ as well as the remaining stages ($s \geq 2$) are solved by ILP. In case that more stages are required than given in $\overrightarrow{\text{eff}}_{\text{min}}$, missing entries are treated as being ∞.

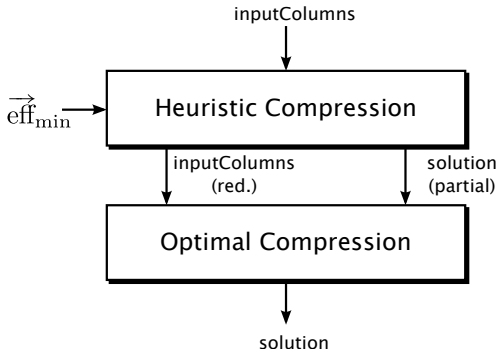


Fig. 3: Proposed flow of heuristic with ILP

5 Experimental Results

Several experiments were performed to compare the proposed optimization methods with the state-of-the-art. We first consider the optimization results based on our cost model using the different methodologies and parameters. A cost model for FPGAs at the HDL-level is typically not perfect as synthesis is not fully predictable in practice, e. g., it is decided during synthesis whether, e. g., LUTs are used for routing or not. In addition, precise timing results can be only obtained from synthesis. Hence, we performed synthesis experiments on generated HDL designs in the last section to check whether the theoretical results can be achieved in practice. In all experiments, the GPCs from Table 1 were used. In addition, we included the corresponding primitive GPCs which lead to a lower output word size. There are exact five cases where primitive GPCs which lower output word size can be obtained from the GPCs in Table 1 that can be used as covering GPCs: $(2, 0, 6; 4)$, $(2, 1, 5; 4)$, $(4, 5; 4)$, $(2, 2, 3; 4)$ and $(1, 2, 5; 4)$ GPCs. For the proposed methods, we included the two possible 4:2 compressor variants as row adders as they provide the highest efficiency combined with the shortest delay.

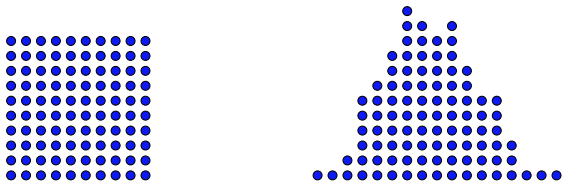
As state-of-the-art compressor tree synthesis algorithms, the heuristic of Parandeh-Afshar et al. [8] and our previous ILP formulation [15] are used. The heuristic of [8] was used with the same efficiency metric as proposed in [13] and used in our heuristic. Therefore, it is equivalent to the simplified heuristic that was published later [13] by the same group (see discussion in Section 3.1). We did not consider a further comparison with the original heuristic of FloPoCo [22] as it was shown that our previous method [15] outperforms this heuristic. All proposed methods as well as our reimplementation of the heuristic of Parandeh-Afshar et al. [8] are available in the open-source arithmetic core generator FloPoCo [23], [25]. It uses the Scalable LP (ScaLP) library [29] which provides a unique interface to different ILP solvers. For the ILP-based methods, the commercial Gurobi [27] ILP solver was selected which can be freely used for academic purposes. The time limit of the ILP solver was set to two hours in all experiments.

5.1 Optimization Quality

In the first experiment, we evaluate the quality of the optimization results using different methodologies. With *quality* we mean the required resources as well as the stage count, where the latter corresponds to the latency (in clock cycles) for pipelined compressor trees and serves as an indicator of the total delay for combinatorial compressor trees. We use two applications as a benchmark for this, a multi-input addition (FloPoCo operator `IntAdderTree`) as well as an x^3 operation (FloPoCo operator `IntPower`). Both represent different shaped dot diagrams as illustrated in Fig. 5. While the multi-input addition is rectangular, the x^3 operation shows a more Gaussian-like distribution of bits, which is more typical for other practical applications like multipliers, polynomials, etc.

In [15] it was demonstrated that multi-input addition problems up to 100 bits can be solved optimally within a time limit of one hour. To be directly comparable to the results presented in [15], we set the number of inputs n identical to their word size which leads to n^2 bits where n was evaluated between 10 and 32. This leads to a quick grow in complexity and corresponds to problem sizes of 100 to 1024 bits. The input word size of the x^3 operation was varied between 6 to 16 bit, leading to problem sizes of 96 to 1586 bits. The optimizations in this experiment were performed for fully pipelined compressor trees, i. e., a register is placed after each GPC, leading to maximum throughput. Hence, the cost of the (1;1) *pseudo* GPC representing a FF in this case was set to 0.5 as for each LUT6, two flip-flops exist on modern FPGAs.

The optimization results for the multi-input addition and x^3 are shown in Fig. 4. Here, the LUT cost in Fig. 4(a)/(b) is the obtained objective value. To allow a comparison, Fig. 4(c)/(d) shows the percentage LUT improvement over Parandeh-Afshar's heuristic [8] while Fig. 4(e)/(f) shows the obtained latencies.



(a) Multi-Input addition of 10 numbers with 10 bit each

(b) x^3 operation for an input word size of 6 bit

Fig. 5: Different dot diagrams shapes of the problems used in the experiments

5.1.1 Comparison of Heuristics

In most of the cases, the proposed heuristic is able to find a solution with less LUTs, leading to LUT reductions of up to 10% for multi-input addition and x^3. In only three out of 34 cases, the heuristic of [8] delivers a better result. However, the main benefit of the proposed heuristic comes from the reduced stage count. In many cases, the heuristic of [8] requires one additional stage which contributes to the total latency or delay. Note that a tree of two-input adders would also require at least four stages for up to 16 inputs (256 bits in Fig. 4(e)) and five stages for up to 32 inputs (1024 bits in Fig. 4(e)).

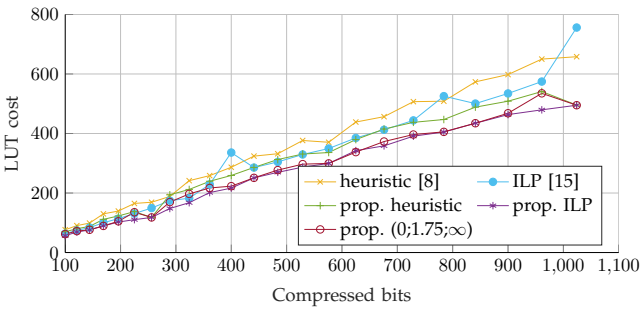
5.1.2 Comparison of ILP-based Methods

One can observe that the ILP method [15] is able to find very good or even optimal results for small problem sizes but fails to find good solutions within the time limit of two hours for larger problems. In contrast to that, the proposed ILP methodology of Section 4.2 is able to find good solutions for much larger problems. For up to three stages, it is always able to find the best solution. For problems requiring four stages and more, the ILP solver always runs into the timeout. Here, it can be observed that the combined approach of Section 4.3 often performs best. Hence, for more than three stages, it is usually beneficial to use the combined method $(0; 1.75; \infty)$, where the first stage is completely solved by the heuristic and stage two is partially solved by using compressors with an efficiency of at least 1.75. As a rule of thumb, problems with up to $S = 3$ stages are best solved by the ILP, while problems with $S > 3$, $S - 3$ stages should pre-solved by the heuristic. The quality can further be fine-tuned by the minimum efficiency parameters.

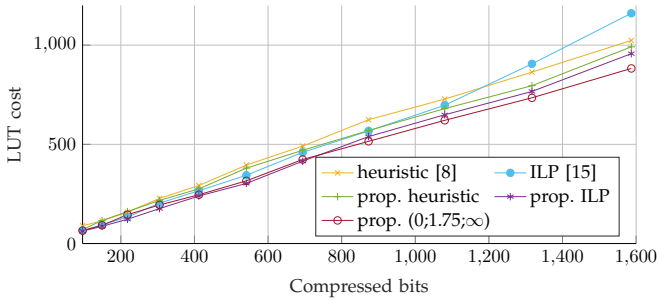
Regarding the stage count, the proposed ILP and the combined methods always find the lowest stage count.

5.1.3 Comparison with Matsunaga's ILP Formulation

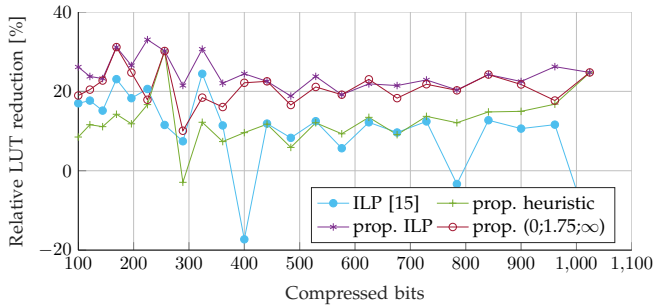
As there is no support for pipelined compressor trees in the ILP formulation of Matsunaga et al. [20], [21], we made a separate experiment using non-pipelined compressor trees for the multi-input addition problems. In Matsunaga's ILP Formulation, all inputs of a GPC must be connected, hence, all primitive GPCs have to be



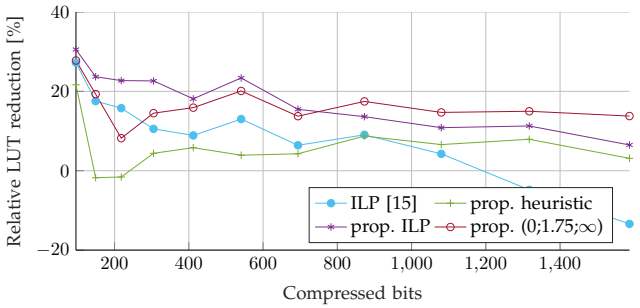
(a) Total LUTs for multi-input addition



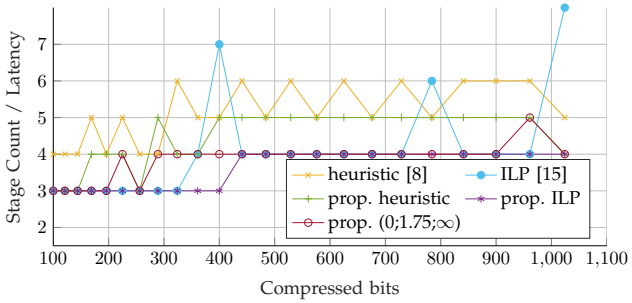
(b) Total LUTs for x^3



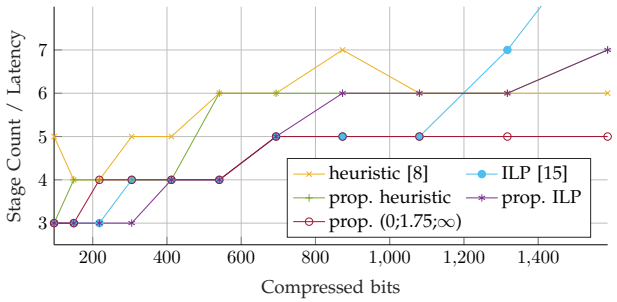
(c) Relative LUT reduction for multi-input addition over Parandeh-Afshar's heuristic [8]



(d) Relative LUT reduction for x^3 over Parandeh-Afshar's heuristic [8]



(e) Latency for multi-input addition



(f) Latency for x^3

Fig. 4: Optimization results as LUT cost and LUT reduction for different methods

obtained. We selected the primitive GPCs by reducing the inputs of the GPCs in Table 1 and the resulting primitive GPC was considered when not violating one of the following rules:

1) if input configuration already exists with less or equal resources
2) if the input column height was all less or equal than one
3) if the LSB input column is zero (e. g., a $(1, 4, 1, 0; 5)$ GPC)
4) if the GPC can be obtained by several other GPCs with equal or less cost (e. g., a $(3, 0, 3; 4)$ GPC, which can be obtained by two (3;2) GPCs)

The output count of the GPCs was reduced when possible. In total, 115 primitive GPCs were obtained.

A detailed comparison of the two methods is given in Table 3. Again, Gurobi was used as ILP solver. Cases where the timeout of two hours was reached are marked with 'TO'. For both methods, only the smallest instances could be solved optimal within the time limit. The proposed ILP always finds solutions in less time (when optimal solution was found within the time limit) or solutions with less or equal LUT cost within the same time.

5.2 Synthesis Results

The compressor trees of the methods of the last section were synthesized together with alternative implementation methods targeting pipelined or combinatorial trees. For that, adder trees of common two-input and ternary adders as well as the advanced linear arrays presented by Hormigo et al. [19] are used. The designs were synthesized for a Virtex 6 (xc6vlx760-ff1760-2) FPGA using Xilinx ISE v13.4 with standard settings for design goal 'speed'. To obtain realistic timing results, input and output registers were added. To not count these registers in the LUT or slice resources, a second run was performed where these registers were placed in the IO blocks of the device (xst option `-iob TRUE`). In addition, the map tool was configured to a minimum *packfactor* (option `-c 1`) to obtain the resource results which yields to a maximum

TABLE 3: Comparison of the proposed ILP with the previous ILP of Matsunaga et al. [20], [21] for multi-input addition using non-pipelined compressor trees (timeouts are marked with 'TO', best results marked bold)

	proposed ILP			Matsunaga's ILP [20], [21]		
word size	Runtime [sec]	LUTs	Stages	Runtime [sec]	LUTs	Stages
10	**19**	**47**	**3**	213	**47**	**3**
11	**30**	**58**	**3**	199	**58**	**3**
12	**30**	**70**	**3**	931	**70**	**3**
13	**66**	**83**	**3**	101	**83**	**3**
14	**181**	**98**	**3**	401	**98**	**3**
15	TO	**115**	**3**	TO	**115**	**3**
16	TO	**132**	**3**	TO	134	**3**
17	TO	**152**	**3**	TO	**152**	4
18	**4923**	**169**	**3**	TO	173	**3**
19	TO	**189**	**4**	TO	191	**4**
20	TO	**209**	**4**	TO	236	5
21	TO	**232**	**4**	TO	234	**4**
22	TO	**256**	**4**	TO	272	**4**
23	TO	**278**	**4**	TO	294	5
24	TO	**311**	**4**	TO	335	5
25	TO	**335**	**4**	TO	355	5
26	TO	**368**	**4**	TO	377	5
27	TO	**399**	**4**	TO	411	5
28	TO	**428**	**4**	TO	441	5
29	TO	**457**	**4**	TO	474	5
30	TO	**493**	**4**	TO	501	5
31	TO	**532**	**4**	TO	541	5
32	TO	**566**	**5**	TO	581	**5**

packing density. This has two reasons. First, all LUTs and FFs are packed in the fewest slices which makes a slice comparison possible. Second, this simulates a device utilization close to 100% which is more realistic and one can figure out if routing congestion may be a problem. All results are given after place&route and for all the designs a valid routing was found.

5.2.1 Pipelined Compressor Trees

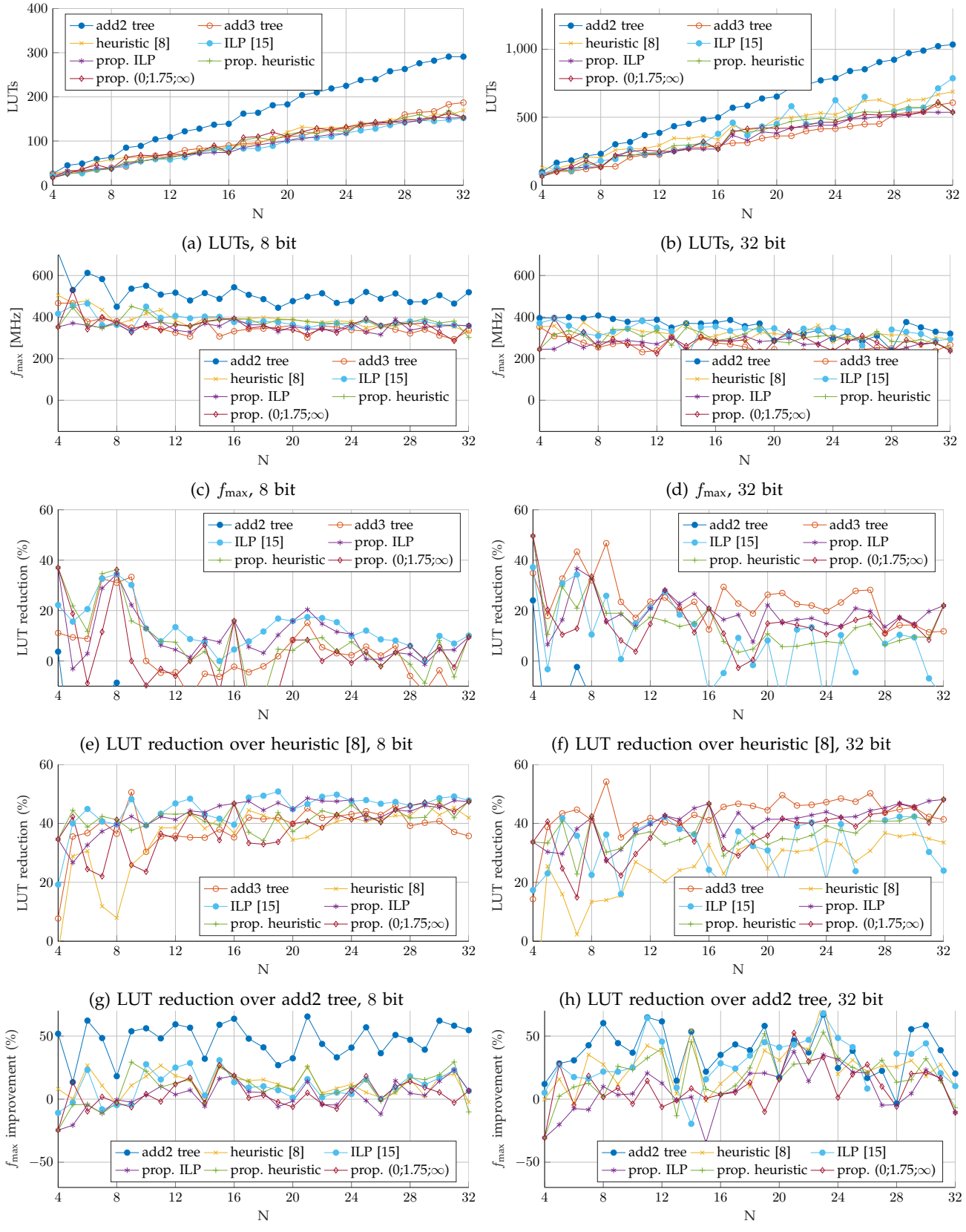
For multi-input addition using pipelined compressor trees, we also implemented a pipelined adder tree using common two-input adders and ternary adders as a baseline (option `method=add2|add3` in the `IntAdderTree` FloPoCo operator). In these, a pipeline register is placed directly after each adder and pipeline balancing registers are included when required (if the input count is not a power-of-two). All designs include the final adder to get a non-redundant result.

The synthesis results of the pipelined compressor trees for the multi-input addition with varying number of input operands and input word sizes of 8 bit and 32 bit are given in Fig. 6. Here, Figs. 6(a)/(b) show the LUT count and Figs. 6(c)/(d) show the maximum clock frequency ($f_{\max}$). The percentage LUT reduction are given in Figs. 6(e)/(f) and (g)/(h) compared to the compressor tree heuristic of [8] and trees of two-input adders, respectively. Figs. 6(i)/(j) show the $f_{\max}$ improvement compared to the ternary adder tree. Average values of the required LUTs and the maximum frequencies as well as the percentage LUT reduction over trees of two-input adders and frequency reductions over ternary adders are given in Table 4. Missing data due to incomplete results where the ILP solver was not able to produce a feasible solution are marked with '–'.

Compared to trees of two-input adders, about 40% of the LUT resources can be saved, on average, by using one of the proposed methods which is a significant improvement to previous work [8] which achieves 35.6% and 24.6% reduction for the 8 bit and 32 bit designs, respectively. The LUT reductions of the proposed methods show a very similar trend like the ternary adder tree. For the 8 bit multi-input addition, the LUT reduction of 43.1% using the proposed ILP is slightly better than the 38.5% using ternary adders while for the 32 bit cases, ternary adders provide the best LUT reduction of 43.2% compared to 40.5% using the proposed ILP.

This experiment confirms that the ternary adder has the highest efficiency (as defined in (2)) but is also the slowest adder type, i. e., the adder trees built from ternary adders are the most compact (in terms of LUTs) but typically provide the lowest $f_{\max}$. The compressor trees typically provide a higher $f_{\max}$ than the ternary adder trees but are still slower than trees of two-input adders. However, the obtained $f_{\max}$ strongly depends on the routing leading to large variations in the $f_{\max}$ improvement. The $f_{\max}$ can be increased by specifying timing constrains. This was not done in this experiment as it comes with the cost of additional LUT resources (used for routing). On average, the compressor trees can be clocked with a 2.1 to 8.8% and 7.0 to 19.0% higher clock frequency than ternary adder trees for the 8 bit and 32 bit designs, respectively. Trees of two-input adders achieve an even higher speed of 46.9% and 37.5% for the 8 bit and 32 bit designs, respectively. Analysis of the timing reports revealed that this difference comes from larger routing delays in compressor trees. Probably, this is due to their irregular nature. However, the compressor trees still provide a very high speed in the order of 300–400 MHz. So, it is likely that the critical path in a larger system lies somewhere else.

For the x^3 operation, the same synthesis experiment was performed but without the adder tree realizations as there is no straight-forward solution to map the adders to an arbitrary dot diagram like shown in Fig. 5(b). They also include a small fraction of resources to compute the partial product bits (using AND gates). The results are plotted in Fig. 7 showing similar trends but slightly less LUT reductions over [8] compared to the multi-input addition. The reason is that the high efficient 4:2 compressor can be less frequently used due to the irregular shape of the dot diagrams. It can be observed that the quality of the proposed methods depend on the problem size. As already observed in Section 5.1, the ILP performs best for small designs (up to 3 stages) while the combined method delivers the best result for larger designs.



(a) LUTs, 8 bit

(b) LUTs, 32 bit

(c) f_{max}, 8 bit

(d) f_{max}, 32 bit

(e) LUT reduction over heuristic [8], 8 bit

(f) LUT reduction over heuristic [8], 32 bit

(g) LUT reduction over add2 tree, 8 bit

(h) LUT reduction over add2 tree, 32 bit

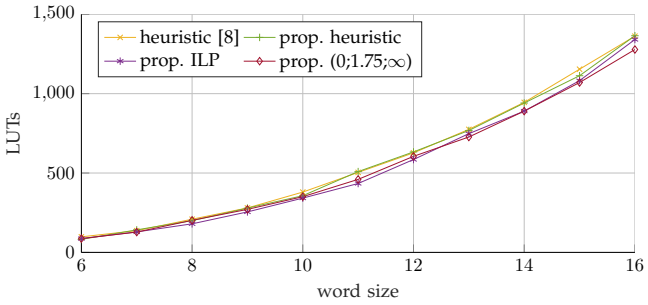
(i) f_{max} imp. over add3 tree, 8 bit

(j) f_{max} imp. over add3 tree, 32 bit

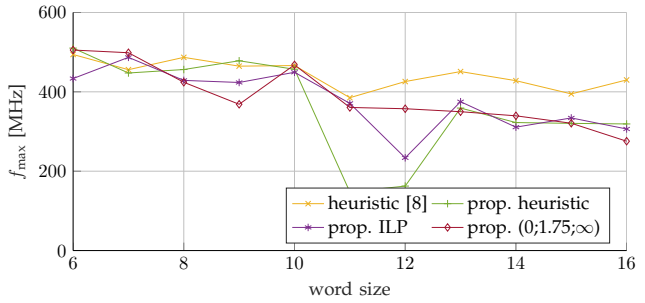
Fig. 6: Synthesis results for pipelined multi-input addition for different methods varying the input operands N for word sizes of 8 bit and 32 bit

TABLE 4: Comparison of average values of LUTs, maximum clock frequency ($f_{\max}$) as well as LUT reduction (comp. to a tree of 2-input adders) and $f_{\max}$ reductions (comp. to a ternary adder tree) for pipelined multi-input addition

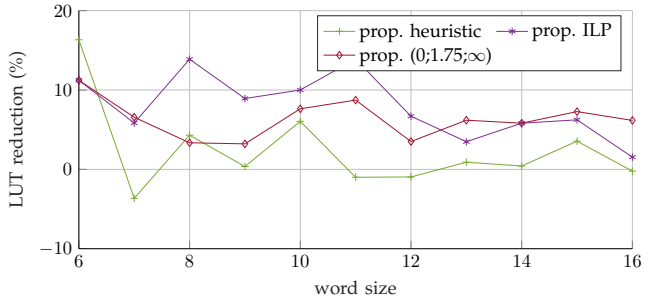
	8 bit				32 bit			
	LUTs	$f_{\max}$ [MHz]	LUT red. (%)	$f_{\max}$ imp. (%)	LUTs	$f_{\max}$ [MHz]	LUT red. (%)	$f_{\max}$ imp. (%)
add2 tree	167.0	512.3	0.0	46.9	590.9	351.7	0.0	37.5
add3 tree	100.0	350.7	38.5	0.0	327.1	258.0	43.2	0.0
heuristic [8]	100.8	390.6	35.6	11.8	418.0	319.5	24.6	25.5
ILP [15]	89.1	382.9	44.9	10.1	–	–	–	–
prop. ILP	92.1	354.2	43.1	2.1	339.5	271.3	40.5	7.0
prop. heuristic	96.2	377.6	41.7	8.8	367.0	302.3	36.7	19.0
prop. (0;1.5;inf)	99.0	361.0	38.4	3.5	358.0	279.7	37.1	9.6



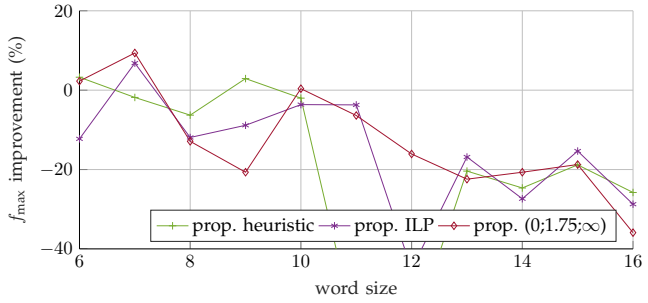
(a) LUTs



(b) $f_{\max}$



(c) LUT reduction over [8]



(d) $f_{\max}$ improvement over [8]

Fig. 7: Synthesis results for pipelined compressor trees computing x^3 for input word sizes from 6 to 16 bit

5.2.2 Combinatorial Compressor Trees

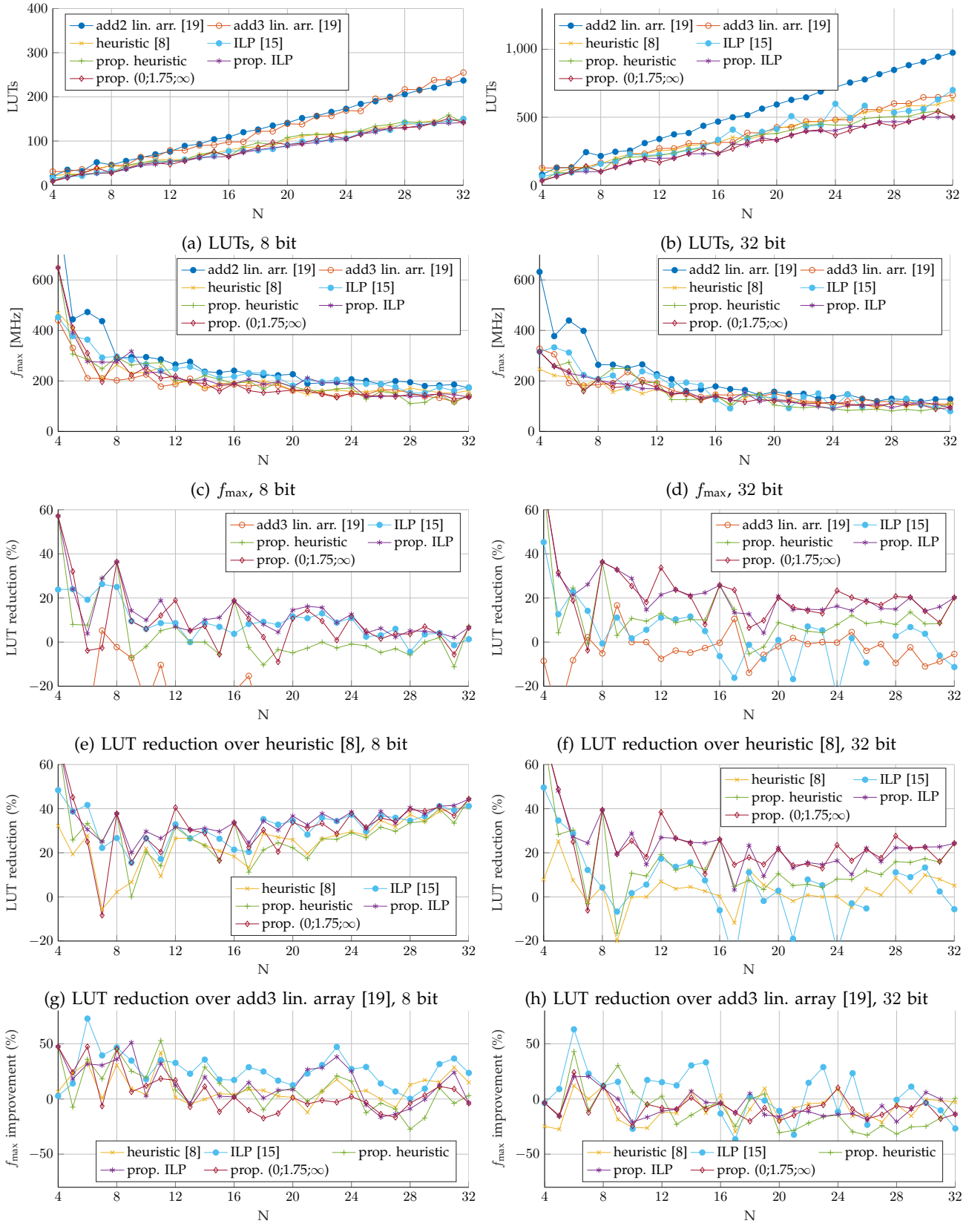
Although less attractive due to the combinatorial routing delays on FPGAs, we performed synthesis experiments with combinatorial compressor trees as applications exist which demand a low latency. In this experiment, no register is used except the input and output registers of the design to obtain realistic timing information including routing. As baseline, the linear arrays presented by Hormigo et al. [19] which are based on two-input adders (add2 lin. array) and ternary adders (add3 lin. array) are used. As the reference VHDL design provided by the authors did not contain the final adders, we also removed the final adders from our designs. The synthesis results are shown in Fig. 8 and average values are listed in Table 5.

The two-input adder linear arrays [19] consume the most resources while providing the highest speed. The ternary adder linear array [19] reduce the average LUT reduction by 2.3% and 24.1% for 8 and 32 bit, respectively, at the cost of a reduced $f_{\max}$. The proposed methods further reduce the average LUTs by 30 ... 40% at a slightly increased $f_{\max}$ for the 8 bit case and a slightly reduced $f_{\max}$ for the 32 bit case.

Of course, the maximum frequency shown in Fig. 8(c)(d) drops much faster compared to the pipelined tree of Fig. 6(c)/(d). However, an important question is: What are the additional costs for the required flip flops? Taking the point $N = 32$, the pipelined compressor tree using the proposed heuristic reaches about 248 MHz while the combinatorial compressor tree can only be clocked with 110 MHz. This 2.3× speedup comes at the cost for additional slices of only 6.6% (136 slices compared to 145). The most of these extra slices are used for the final adder in the pipelined compressor tree. Hence, pipelined compressor trees should be used whenever the latency requirements allows.

6 Conclusion

It was demonstrated that there was still a lot of room for improvement in the design of compressor trees on FPGAs which can be exploited by the proposed methods. To do so, an improved heuristic and an optimal solution as well as a combination of both were presented. All algorithms support GPCs of arbitrary shape as well as row adders like common 2-input adders, ternary adders or 4:2 compressors. The heuristic is typically very fast



(a) LUTs, 8 bit

(b) LUTs, 32 bit

(c) f_{max}, 8 bit

(d) f_{max}, 32 bit

(e) LUT reduction over heuristic [8], 8 bit

(f) LUT reduction over heuristic [8], 32 bit

(g) LUT reduction over add3 lin. array [19], 8 bit

(h) LUT reduction over add3 lin. array [19], 32 bit

(i) f_{max} imp. over add3 lin. array [19], 8 bit

(j) f_{max} imp. over add3 lin. array [19], 32 bit

Fig. 8: Synthesis results for combinatorial multi-input addition for different methods varying the input operands N for word sizes of 8 bit and 32 bit

TABLE 5: Comparison of average values of LUTs, maximum clock frequency ($f_{\max}$) as well as LUT reduction (comp. to a tree of 2-input adders) and $f_{\max}$ reductions (comp. to a ternary adder tree) for combinatorial multi-input addition

	8 bit				32 bit			
	LUTs	$f_{\max}$ [MHz]	LUT red. (%)	$f_{\max}$ imp. (%)	LUTs	$f_{\max}$ [MHz]	LUT red. (%)	$f_{\max}$ imp. (%)
add2 lin. arr. [19]	127.5	270.3	0.0	41.0	530.1	208.8	0.0	26.9
add3 lin. arr. [19]	124.6	186.7	2.3	0.0	375.6	158.3	24.1	0.0
heuristic [8]	88.4	205.0	27.2	9.8	364.1	141.5	26.9	−8.9
ILP [15]	82.5	233.2	34.4	25.9	–	–	–	–
prop. ILP	88.1	209.6	31.0	9.8	326.0	145.4	37.7	−10.6
prop. heuristic	79.7	217.9	37.5	14.5	296.6	148.3	44.1	−6.6
prop. (0;1.75;∞)	82.6	202.0	34.9	4.9	295.6	145.8	43.9	−8.0

(within seconds) and delivers solutions with adequate quality while the optimal ILP-based solution is naturally limited to small problem sizes. To further enhance the optimization quality, a combined optimization was proposed where the first stages are either completely or partially solved by the heuristic and the remaining problem is solved by ILP. It was shown that this strategy fits well to large size problems. Non-optimal decisions in the first stages seem to have little impact on the overall quality. It turned out that the combined optimization approach should be used for large problems with a stage count of more than three.

All methods were extensively compared to state-of-the-art compressor tree methods [8], [15], [19]–[21] showing significant resource reductions in most of the cases. The proposed methods provide average LUT reductions of about 40% compared to trees of 2-input adders but are about 12…20% slower. They show a similar LUT reduction like ternary adder trees but are typically faster. For pipelined compressor trees, the proposed methods additionally find a lower or equal latency than the previous heuristic [8] or the trees of two-input adders. For combinatorial compressor trees, LUT reductions of about 40% can be achieved. However, due to the efficient use of the fast carry chain, the ternary adder linear arrays [19] perform faster in several cases.

The comparison between pipelined and combinatorial compressor trees clearly shows that a) pipelining is necessary to achieve state-of-the-art performance and b) the FFs cost for pipelining are nearly hidden in the used slice resources and, thus, pipelining is very inexpensive on FPGAs. Hence, as long as the latency allows, pipelining should be used.

All methods are available as open source within the FloPoCo arithmetic core generator [23] to improve the existing arithmetic cores and to obtain a full reproducibility of our results.

7 Acknowledgement

The authors would like to thank Dr. Javier Hormigo for providing the sources for the adder trees proposed in [19] and for his very kind support in reproducing the results.

References

[1] E. E. Swartzlander, "Merged Arithmetic," *submitted to IEEE Transactions on Computers and currently under review*, vol. C-29, no. 10, pp. 946–950, Oct. 1980.

[2] K. A. Feiste and E. E. Swartzlander, "Merged arithmetic revisited," *Signal Processing Systems, 1997. SIPS 97 - Design and Implementation., 1997 IEEE Workshop on*, pp. 212–221, 1997.

[3] T. Kim and J. Um, "A practical approach to the synthesis of arithmetic circuits using carry-save-adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 5, pp. 615–624, 2000.

[4] J. Um and T. Kim, "An optimal allocation of carry-save-adders in arithmetic circuits," *submitted to IEEE Transactions on Computers and currently under review*, vol. 50, no. 3, pp. 215–233, Mar. 2001.

[5] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," *ICCAD 2004. International Conference on Computer Aided Design*, pp. 791–798, 2004.

[6] C. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, no. 1, pp. 14–17, 1964.

[7] L. Dadda, "Some Schemes For Parallel Multipliers," *Alta Frequenza*, vol. 45, no. 5, pp. 349–356, 1965.

[8] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient Synthesis of Compressor Trees on FPGAs," in *Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2008, pp. 138–143.

[9] ——, "Improving Synthesis of Compressor Trees on FPGAs via Integer Linear Programming," in *Design, Automation and Test in Europe (DATE)*. IEEE, 2008, pp. 1256–1261.

[10] A. R. Meo, "Arithmetic Networks and Their Minimization Using a New Line of Elementary Units," *submitted to IEEE Transactions on Computers and currently under review*, vol. C-24, no. 3, pp. 258–280, 1975.

[11] K. A. C. Bickerstaff, M. Schulte, and E. E. Swartzlander, "Reduced area multipliers," in *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, 1993, pp. 478–489.

[12] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting Fast Carry-Chains of FPGAs for Designing Compressor Trees," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 242–249.

[13] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor Tree Synthesis on Commercial High-Performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–19, Dec. 2011.

[14] M. Kumm and P. Zipf, "Efficient High Speed Compression Trees on Xilinx FPGAs," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014, pp. 171–182.

[15] ——, "Pipelined Compressor Tree Optimization Using Integer Linear Programming," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–8.

[16] T. B. Preußer, "Generic and Universal Parallel Matrix Summation with a Flexible Compression Goal for Xilinx FPGAs," in *Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7.

[17] M. Ortiz, F. Quiles, J. Hormigo, F. J. Jaime, J. Villalba, and E. L. Zapata, "Efficient Implementation of Carry-Save Adders in FPGAs," in *IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, 2009, pp. 207–210.

[18] W. Kamp, A. Bainbridge-Smith, and M. Hayes, "Efficient Implementation of Fast Redundant Number Adders for Long Word-Lengths in FPGAs," in *2009 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2009, pp. 239–246.

[19] J. Hormigo, J. Villalba, and E. L. Zapata, "Multioperand Redundant Adders on FPGAs," *submitted to IEEE Transactions on Computers and currently under review*, vol. 62, no. 10, pp. 2013–2025, 2013.
[20] T. Matsunaga, S. Kimura, and Y. Matsunaga, "An Exact Approach for GPC-Based Compressor Tree Synthesis," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96-A, no. 12, pp. 2553–2560, Dec. 2013.
[21] ——, "Power and Delay Aware Synthesis of Multi-Operand Adders Targeting LUT-Based FPGAs," *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 217–222, 2011.
[22] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic Core Generation Using Bit Heaps," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2013, pp. 1–8.
[23] 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.
[24] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.
[25] F. de Dinechin. FloPoCo Project Website. [Online]. Available: http://flopoco.gforge.inria.fr
[26] M. Kumm, M. Hardieck, J. Willkomm, P. Zipf, and U. Meyer-Baese, "Multiple Constant Multiplication with Ternary Adders," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2013, pp. 1–8.
[27] Gurobi Optimization Inc. Gurobi Website. [Online]. Available: http://www.gurobi.com
[28] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 3.2," Takustr.7, 14195 Berlin, Tech. Rep., 2016.
[29] M. Kumm, T. Schönwälder, P. Sittel, and P. Zipf. (2017) Scalable Linear Programming (ScaLP) Project Website. [Online]. Available: http://www.uni-kassel.de/go/scalp

Martin Kumm received the Dipl.-Ing. degree in electrical engineering from Fulda University of Applied Sciences, Fulda, and the Technical University of Darmstadt, Germany, in 2003 and 2007, respectively. In 2015 he received his Ph.D. (Dr.- Ing.) degree from the University of Kassel, Germany. From 2003 to 2009, he was with GSI Darmstadt, working on digital RF control systems for particle accelerators. His current research interests in the Digital Technology Group of the University of Kassel are arithmetic circuits and their optimization in the context of reconfigurable systems.

Johannes Kappauf received the B.Sc. degree in computer science from University of Kassel, Kassel, Germany in 2015. Since 2016 he is working as a research assistant in the Digital Technology group at University of Kassel. He is currently working toward the M.Sc. degree in computer science at University of Kassel. His main interests include digital signal processing and arithmetic circuits.