Pipelined Compressor Tree Optimization using Integer Linear Programming

Martin Kumm and Peter Zipf
University of Kassel, Germany
Digital Technology Group
Email: {kumm,zipf}@uni-kassel.de

***Abstract*—Compressor trees offer an effective realization of the multiple input addition needed by many arithmetic operations. However, mapping the commonly used carry save adders (CSA) of classical compressor trees to FPGAs suffers from a poor resource utilization. This can be enhanced by using generalized performance counters (GPCs). Prior work has shown that high efficient GPCs can be constructed by exploiting the low-level structure of the FPGA. However, due to their irregular shape, the selection of those is not straight forward. Furthermore, the compressor tree has to be pipelined to achieve the potential FPGA performance. Then, a selection between registered GPCs or flip-flops has to be done to balance the pipeline. This work defines the pipelined compressor tree synthesis as an optimization problem and proposes a (resource) optimal method using integer linear programming (ILP). Besides that, two new GPC mappings with high efficiency are proposed for Xilinx FPGAs.**

I. Introduction

Many arithmetic operations require the summation of partial results. These include multiplications (real and complex), polynomials (e. g., for function approximation), digital filters and linear transforms, just to name a few. The problem of building fast and compact multiple input adders has a long history and was motivated by the design of fast parallel multipliers. Compressor trees as introduced by Wallace [1] and refined by Dadda [2] in the 1960'th are still the fastest and most efficient circuit structure which can be found in the arithmetic units of today's CPUs together with Booth's encoding technique. The main idea is the avoidance of long carry propagations by 'saving' the carry-out of the full-adders for the next stage, leading to the carry save adder (CSA). However, the situation on FPGAs is much different. With the introduction of fast carry chains on FPGAs, the ripple carry adder (RCA) became so fast that CSA based compressor trees became worse as the short carry propagation was shattered by the large routing delays.

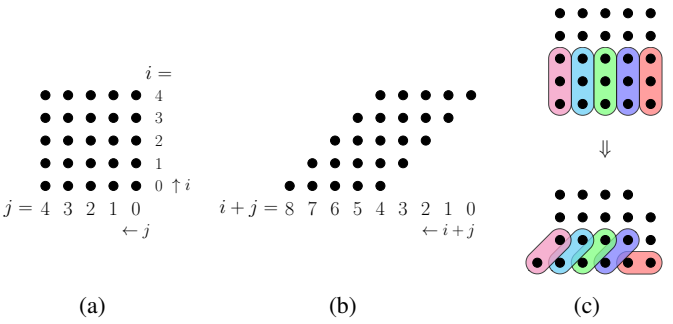
However, it was demonstrated in the last years that the usage of generalized parallel counters (GPCs) as replacement of full-adders can reduce the combinatorial delay in compressor trees while having a similar resource consumption [3]–[8]. A full-adder (also known as 3:2 counter) basically counts the number of input bits which are one. GPCs [9] (also called multicolumn counters [10]) allow that input bits may have different weights. A GPC is commonly denoted as tuple $(p_{k-1}, p_{k-2}, \ldots, p_0; q)$, 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 thus represented by a 4 bit number.

Optimization methods for delay-optimized combinatorial compressor trees using GPCs for FPGAs were proposed by Parandeh-Afshar et al. [3]–[5]. They proposed a heuristic [3] and an integer linear programming (ILP) method [4] for compressor tree synthesis using look-up table (LUT) based GPCs. Their ILP method can only be applied to small problems [5], so the heuristic was extended in a later work where also parts of the FPGA carry chain were considered for the GPCs [5]. Their heuristic focuses on a good utilization of GPCs having a high compression ratio (the ration between input and output bits) to reduce the number of stages and, thereby, the delay. 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, which are supported on modern FPGAs.

An ILP formulation targeting the power and delay optimization of compressor trees was proposed recently by Matsunaga et al. [6], [8]. Here, power and delay are minimized by minimizing the number of compression stages as well as the number of GPCs. Their ILP formulation is given for LUT-based GPCs of input size 6 and output word size 3 but should be easily extendable to different GPCs. They could reduce the runtime of a previous ILP [4] and showed that up to 28% GPC reductions can be achieved compared to previous heuristics [3], [7] by using an optimal ILP approach.

A versatile method for representing compressor trees in arithmetic core generation as a data structure which is called a 'bit heap' was proposed recently by de Dinechin et al. [11]. It is a direct representation of dot diagrams [10] (see Fig. 1(a)) which yields to a great abstraction from a software engineering point of view. They also analyzed different compressors for their efficiency and use the most efficient ones in their heuristic instead of forcing a good utilization. One interesting result from their efficiency analysis is that the ternary adder has the best efficiency (implied that all of the inputs can be used).

However, it was shown by us that ternary adders are slow compared to common two-input adders on Xilinx FPGAs [12]. While the performance penalty compared to a two-input adder is up to 10% for Altera FPGAs, it is close to 50% for Xilinx FPGAs which makes the ternary adder unattractive for high speed applications on Xilinx FPGAs. Hence, we searched for



(a) (b) (c)

Fig. 1: Example dot diagrams: (a) 5-input addition; (b) 5×5 multiplier; (c) compression of a 5-input adder using full-adders

high efficient compressors by exploiting the low-level structure of Xilinx FPGAs and found novel GPCs as well as a fast 4:2 compressor [13].

The relatively large routing delays of FPGAs usually require the use of pipelining to reach the throughput capabilities of the FPGA. It was shown in another adder-based arithmetic circuit for constant multiplication that it is important to use the output flip-flops located in the Slice/ALM which contain logic instead of adding new Slice/ALM resources to add required flip-flops to balance the pipeline [14]. Thus, the focus and the main contribution of this paper are a novel ILP-based optimization method for the pipelined compressor tree synthesis on FPGAs. Besides that, more efficient GPCs for Xilinx FPGAs are proposed. In the ILP formulation, we are not trying to reduce the overall combinatorial delay like in previous work but to use a resource-minimal pipelining for a throughput increase of multiple times. For that, we assume that each compressor stage is pipelined which leads to a critical path consisting of a local routing, one layer of logic and a short carry chain.

II. Compressor Tree Synthesis

The synthesis of compressor trees is explained by introducing the dot-representation of the compression problem. Take for example the simple unsigned multiple-input addition, represented as

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

where each X_i represents a bit vector with $x_{i,j}$ denoting a single bit of weight j. The compression problem can be represented in a dot-diagram, where each bit which has to be added is represented as a single dot. An example dot-diagram where five bit vectors of five bits each have to be added is shown in Fig. 1(a). By convention, the rightmost dot represents weight zero with increasing weight to the left. The order of the bits within a column does not matter. Another example is the multiplication of two bit vectors

$$X \cdot Y = \sum_{i,j} 2^{i+j} x_i y_j \ , \tag{2}$$

where x_i and y_i represent the bits of X and Y, respectively. The corresponding dot diagram is shown in Fig. 1(b).

TABLE I: Dot transformations for various compressors

Compressing element	Dot transformation	Compressing element	Dot transformation
$(3,2)$ counter (full-adder)		$(6;3)$ GPC	
$(6,0,6;5)$ GPC		$(1,4,1,5;5)$ GPC	
$(1,4,0,6;5)$ GPC		$(2,0,4,5;5)$ GPC	
ternary adder		4:2 compressor	

The synthesis of a compressor tree can now be obtained by applying compressors to the dot diagram. With 'compressor' we denote any circuit which is able to reduce the number of bits. This can be a single column counter like the (3,2) full-adder, a GPC like the $(3,5;3)$ GPC or an $N : M$ compressor [10] like the $4:2$ compressor mentioned before. Each compressor can be seen as a transformation in the dot diagram. The full-adder for example can replace three dots in the same column (two inputs and carry-in) by a row of two dots (the sum and carry-out). Hence, one dot is removed with each full-adder. Transformations for various compressors are shown in Table I. An example of the first compression stage of the 5-input adder tree of Fig. 1(a) using full-adders is shown in Fig. 1(c). Bits that are compressed or generated by the same full-adder are highlighted with the same color. The 25 bit are reduced to 20 bit. This procedure is repeated until at most two or three rows remain; these are then compressed to a single row using a ripple-carry adder or a ternary adder, respectively. This last adder is commonly called the vector merge adder (VMA).

It can be observed that only the number of bits in each column matters. Therefore, we introduce a more compact tabular representation for the compression process which is also beneficial for later explanation of the ILP optimization. The tabular representation of the complete compression of the 5-input adder example is shown in Fig. 2. Each column in the tabular representation corresponds to the bit weight

		5	5	5	5	5	bits in stage 0
−						3	} (3;2) counter
+					1	1	}
−					3		} (3;2) counter
+				1	1		}
−				3			} (3;2) counter
+			1	1			}
−			3				} (3;2) counter
+		1	1				}
−		3					} (3;2) counter
+	1	1					}
=	1	4	4	4	4	3	bits in stage 1
	1	4	4	4	4	3	bits in stage 1
−						3	} (3;2) counter
+					1	1	}
−					3		} (3;2) counter
+				1	1		}
−				3			} (3;2) counter
+			1	1			}
−			3				} (3;2) counter
+		1	1				}
−		3					} (3;2) counter
+	1	1					}
=	1	3	3	3	3	1	bits in stage 2
	1	3	3	3	3	1	bits in stage 2
−					3		} (3;2) counter
+				1	1		}
−				3			} (3;2) counter
+			1	1			}
−			3				} (3;2) counter
+		1	1				}
−		3					} (3;2) counter
+	1	1					}
=	2	2	2	2	1	1	bits in stage 3

Fig. 2: Tabular representation of the compression of the 5-bit, 5-input adder using full-adders

(increasing from right to left as in the dot diagram). Each numeric entry corresponds to the number of bits of equal weight. Each compressor removes and adds bits from specific columns which is denoted by rows starting with a '$-$' or '$+$', respectively. After one stage of compression the sum of each column is computed (rows starting with a '$=$'), which is taken as the input of the next compression stage. Hence, our example can be compressed to two rows by using 14 full-adders, distributed over three stages. Note that only connected inputs of the compressors have to be subtracted (see, e. g., the $(6, 0, 6; 5)$ GPC in Fig. 4 where only five inputs are used each).

III. FPGA Specific Compression

Recent work on FPGA compressor trees revealed that an important metric for FPGA compressors is the efficiency [11] which is defined as the number of removed bits, given by the difference between the input bits b_i and output bits b_o, denoted $\delta = b_i - b_o$, in relation to the number of the LUTs needed (k):

$$E = \frac{\delta}{k} \tag{3}$$

It turned out that previous compressors like the GPCs mapped to LUTs and carry-chains [5] as well as LUT-based GPCs have an efficiency of at most one which is identical to the efficiency of a two-input adder [11]. Optimized GPC mappings led to efficiencies of up to 1.5 [11], [13]. The ternary adder as well as the $4:2$ compressor have an efficiency of $E = 2 - \frac{2}{k}$,

TABLE II: Comparison of different compression elements

GPC / Compressor	#LUT6 (k)	Efficiency ($E = \delta/k$)	delay
LUT based GPCs from [11]			
(3;2) GPC	1	1	$\tau_{\mathrm{L}} \approx \tau$
(6;3) GPC	3	1	$\tau_{\mathrm{L}} \approx \tau$
(1,5;3) GPC	3	1	$\tau_{\mathrm{L}} \approx \tau$
Improved GPC mappings from [13], originally introduced in [5]:			
(6;3) GPC	3	1	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + 3\tau_{\mathrm{CC}} \approx 3\tau$
(1,5;3) GPC	2	1.5	$\tau_{\mathrm{L}} + 2\tau_{\mathrm{CC}} \approx \tau$
(2,3;3) GPC	2	1	$\tau_{\mathrm{L}} + 2\tau_{\mathrm{CC}} \approx \tau$
(7;3) GPC	3	1.33	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + 3\tau_{\mathrm{CC}} \approx 3\tau$
(5,3;4) GPC	3	1.33	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + 3\tau_{\mathrm{CC}} \approx 3\tau$
(6,2;4) GPC	3	1.33	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + 3\tau_{\mathrm{CC}} \approx 3\tau$
GPCs and 4:2 compressor from [13]:			
(5,0,6;5) GPC	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(1,4,1,5;5) GPC	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(1,4,0,6;5) GPC	4	1.5	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(2,0,4,5;5) GPC	4	1.5	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + 4\tau_{\mathrm{CC}} \approx 3\tau$
4:2 compressor	k	$2 - \frac{2}{k}$	$\tau_{\mathrm{L}} + k\tau_{\mathrm{CC}}$
Adder with k BLE:			
2-input adder	k	1	$\tau_{\mathrm{L}} + k\tau_{\mathrm{CC}}$
3-input adder	k	$2 - \frac{2}{k}$	$2\tau_{\mathrm{L}} + \tau_{\mathrm{R}} + k\tau_{\mathrm{CC}} \approx 3\tau + k\tau_{\mathrm{CC}}$
Proposed GPCs:			
(6,0,6;5) GPC	4	1.75	$\tau_{\mathrm{L}} + 4\tau_{\mathrm{CC}} \approx \tau$
(1,3,2,5;5) GPC	4	1.5	$\tau_{\mathrm{L}} \approx \tau$

where k is the number of LUTs which is equal to the number of input columns [11], [13]. Hence, their efficiency is getting close to two for large values of k. The list of compressors used in this work and their properties (LUT6 cost, efficiency and delay) are summarized in Table II. The delays are denoted by the symbols τ_{L}, τ_{CC} and τ_{R} and correspond to the LUT, one bit of carry propagation and a local routing, respectively. As $\tau_{\mathrm{L}} \approx \tau_{\mathrm{R}}$ (typically 0.3 ns for the Virtex 6 family) while $\tau_{\mathrm{CC}} \ll \tau$ (typically 15 ps), an approximate delay is given with $\tau \approx \tau_{\mathrm{L}} \approx \tau_{\mathrm{R}}$. The upper part of the table was published earlier [11], [13] and is reproduced here for the sake of comparison. The lower part of the table lists the properties of the two GPCs proposed in this work (see Section V).

Applying these highly efficient compressors leads to two FPGA related problems: First, the 'shapes' of most of the high efficient GPCs are highly irregular, i. e., the number of input bits per column is varying (from zero to six) as illustrated in the middle part of Table I. Hence, a method is needed to find an efficient covering even for these irregular GPCs. Second, the delay becomes pretty high only after a few stages of compression due to the large routing delays of FPGAs. The common countermeasure is using pipelining. There, the schedule of pipeline stages is a crucial issue as a lot of additional pipeline registers may be needed. FPGA-based compressors can be typically pipelined without extra resources as the (otherwise unused) flip-flops of the slice (Xilinx) or ALM (Altera) are used. Thus, for stages in which a pipeline stage is foreseen, it may be better to cover more bits with (possibly underutilized) GPCs instead of covering bits with

				5	5	5	5	5	bits in stage 0
−					1	4	1	5	} (1,4,1,5;5) GPC
+				1	1	1	1	1	}
−				1	4	1	4		} (1,4,1,5;5) GPC
+			1	1	1	1	1		}
=			1	6	2	2	2	1	bits in stage 1
			1	6	2	2	2	1	bits in stage 1
−				6					} (6;3) GPC
+		1	1	1					}
=		1	2	1	2	2	2	1	bits in stage 2

Fig. 3: Tabular representation of the compression of the 5-bit, 5-input adder using GPCs

				5	5	5	5	5	bits in stage 0
−					2	0	4	5	} (2,0,4,5;5) GPC
+				1	1	1	1	1	}
−				5	0	5			} (6,0,6;5) GPC
+		1	1	1	1	1			}
−					3		1		} 4 FF for pipeline balancing
+					3		1		}
=		1	1	2	5	2	2	1	bits in stage 1
		1	1	2	5	2	2	1	bits in stage 1
−		1	1	2	5				} (1,3,2,5;5) GPC
+	1	1	1	1	1				}
−					3	2	2	1	} 5 FF for pipeline balancing
+					3	2	2	1	}
=	1	1	1	1	1	2	2	1	bits in stage 2

Fig. 4: Tabular representation of the pipelined compression of the 5-bit, 5-input adder using pipelined GPCs and flip-flops

additional flip-flops to balance the pipeline.

The first problem is illustrated in Fig. 3, where the same problem as in Fig. 2 was optimized using FPGA-specific GPCs. Only three GPCs are needed which are distributed over two stages. The total cost for compression is 11 (4+4+3) according to the LUT6 cost in Table II. The full-adder solution of Fig. 2 would consume 14 LUT6 which are distributed over three stages, leading to a delay increase of about 50%. The second problem is illustrated in Fig. 4 where the pipelined compression is considered. Each GPC now includes a flip-flop at the output. Additional flip-flops (for pipeline balancing) are now also considered as compressors even though they do not compress anything. The cost of each flip-flop was set to 0.5 as for each LUT6, two flip-flops exist on modern FPGAs. A complete covering of compressor inputs (incl. flip-flops) is necessary to obtain a valid pipeline. This can be easily checked by subtracting the bits of the first row of each stage from the number of compressor inputs which has to be less-or-equal-than zero for each column. In this example, three registered GPCs of cost 12 (4+4+4) are used and nine additional flip-flops (cost 4.5) to balance the pipeline leading to a total cost of 16.5. Pipelining the full-adder solution in Fig. 2 would require 15 additional flip-flops and a total cost of 21.5.

IV. ILP Formulation

A. Basic Formulation

An integer linear programming (ILP) formulation for the problems described above (compressor tree synthesis with or without pipelining) is proposed in this section. It can be solved by any standard ILP solver like the commercial CPLEX [15] or the open-source tool SCIP [16].

Similar to previous work [6], [8], the optimization is performed on a set of integer variables $k_{s,e,c}$ which denote the number of compressors of type $e = 0 \dots E-1$ in stage $s = 0 \dots S-1$ and column $c = 0 \dots C-1$. The used compressors are target dependent (number of LUT inputs and carry-chain layout). Each 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. If compressor $e = 4$ is a $(1, 5; 3)$ GPC, then $M_{4,0} = 5$, $M_{4,1} = 1$ and $K_{4,0\dots2} = 1$. With this notation, also compressors with more that one output bit per column can be represented. Furthermore, a cost value c_e is assigned to each of the compressors as a real number.

One key element in the model is that we always include a single-input, single-output compressor in the set of compressors. In pipelined compressor trees, this represents a flip-flop with real cost, in combinatorial compressor trees it represents a wire with zero cost. With this, we can simply constraint a complete coverage of input bits by compressors without the need of a special treatment for wires. Of course, a mixture of both can be applied by adjusting the flip-flop/wire cost for each stage, e. g., if only every 2nd stage should be pipelined.

Another set of integer variables $N_{s,c}$ is used which represents the number of bits in column c and stage s. Clearly, the variables $N_{0,1\dots C-1}$ correspond to the number of input bits (first line in the tabular representation) and are hence constant. At the output stage the number of bits in each column must be less than the number of input rows of the VMA (typically 2 or 3). The stage limit S is chosen as large as necessary but the actual stage count may be much lower than S. To account for that, an additional binary variable is introduced for each stage:

$$D_s = \begin{cases} 1 & \text{if the output stage is equal to } s \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The problem can now be stated as follows:

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

subject to

$$\text{C1: } N_{s-1,c} \le \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} M_{e,c+c'}\, k_{s-1,e,c+c'} \quad \left.\begin{array}{l} s = 1 \dots S-1, \\ c = 0 \dots C-1, \\ \text{if } D_s = 0 \end{array}\right.$$

$$\text{C2: } \quad N_{s,c} = \sum_{e=0}^{E-1} \sum_{c'=0}^{C_e-1} K_{e,c+c'}\, k_{s-1,e,c+c'} \quad \left.\begin{array}{l} s = 1 \dots S-1, \\ c = 0 \dots C-1 \end{array}\right.$$

$$\text{C3: } \quad N_{s,c} \le \begin{cases} 2 & \text{for two-input VMA} \\ 3 & \text{for ternary VMA} \end{cases} \quad \text{if } D_s = 1$$

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

The first constraint (C1) ensures that all bits in each column and stage except the output stage are connected to inputs of compressors (C_e provides the number of columns for compressor e). Constraint C2 is used to compute the number of bits produced by compressors which is taken as input to the next stage. The column height of the output stage is constrained by C3. Constraint C4 simply ensures that there is exactly one stage that is marked as the output stage (only one of the D_s variables has to be set to true).

So far, the formulation above is not linear as there are 'if-conditions' in constraints C1 and C3. They can be linearized by the following constraints:

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

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

Here, constant I has to be set to a sufficiently large positive number such that $I > N_{s,c}$ (for any s and c). Then, C1 and C3 get unconstrained in case of $D_s = 1$ and $D_s = 0$, respectively.

Note that the ILP formulation was independently developed but contains similar ideas compared to the ILP formulation of Matsunaga et al. [6], [8]. There, integer variables are also used to count the number of GPCs per column and stage. However, the presented formulation is more general as it is capable of using any GPC (including target specific ones) with even more than one output bit per column, and, the most important difference, it supports the optimization of pipelined compressor trees.

B. Upper Bound for the Stage Count

The minimum number of stages greatly depends on the compression ratio (b_i/b_o). The higher the ratio, the lower the depth. For single column counters and GPCs with equal column heights, the minimal number of stages can be directly computed [9]. For a mixture of different compressors with unequal height like for the given problem this is not that trivial. As we only have to specify an upper bound on the number of stages (S), we use the minimum stage count of a full-adder compressor tree. This can be obtained from Dadda's recursion formula, which gives the maximum column height in stage s

$$h_{s+1} = \left\lceil \frac{3}{2} h_s \right\rceil \ , \tag{5}$$

starting from $h_0 = 2$, leading to the sequence $\{2, 3, 4, 6, 9, 13, 19, 28, 42, 63, ...\}$. With this rather pessimistic bound, we can always guarantee that a solution is found within S stages.

C. Limiting the Stage Count

An important property is the delay in combinatorial compressor trees and the latency in pipelined compressor trees. Although the stage count will be low due to the cost function, it may be higher than needed. Of course, a fixed depth can be achieved by setting $D_{S'} = 1$ for a required S' and checking if the solution is feasible. A more elegant way to force the minimum possible stage count is by modifying the objective as follows:

$$\text{minimize} \sum_{s=0}^{S-1} \left(\sum_{c=0}^{C-1} \sum_{e=0}^{E-1} c_e k_{e,c} + sLD_s \right)$$

Again, L must be a sufficiently large constant (larger than the total number of LUTs in the compressor tree). Take, e. g., $L = 1000$, if there exist feasible solutions with 80 LUTs using four stages and 100 LUTs using three stages, the resulting objective value will be 4080 and 3100 for $D_4 = 1$ and $D_3 = 1$, respectively, and the lower depth will be preferred.

D. Support for Variable Column Compressors

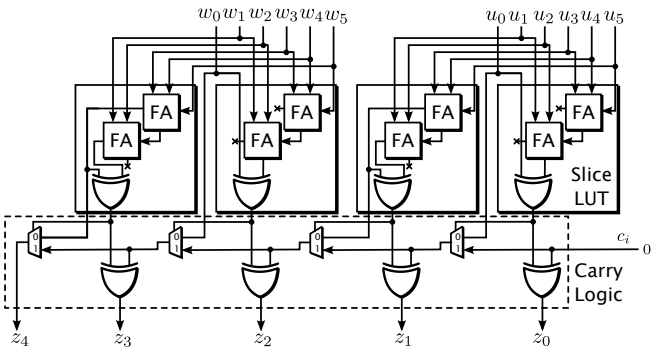
The considered compressors so far had a fixed shape. However, there exist compressors with varying width like the common two-input adder, the ternary adder or the $4:2$ compressor as shown in the lower part of Table I. Of course, one can model n different compressors for n different column sizes but this will blow up the number of variables and, thus, the optimization time. 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 procedure is demonstrated for the example of a ternary adder. The ternary adder can be divided into three partial compressors: e_L (the lowest column), e_M (multiple columns in the middle) and e_H (the highest column). The lowest (rightmost) column compressor reduces four bit of the first column (three plus carry-in) into one bit ($M_{e_L,0} = 4$ and $K_{e_L,0} = 1$, see Table I). The middle column compressors reduce three bits to one bit of the same column ($M_{e_M,0} = 3$ and $K_{e_M,0} = 1$). The highest (leftmost) compressor has a single input bit and produces two output bits in two columns ($M_{e_H,0} = 2$ and $K_{e_H,0\ldots1} = 1$). 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 following constraints:

$$\left.\begin{aligned} \text{C5:} \quad & k_{s,e_M,c+1} = k_{s,e_L,c} \\ \text{C6:} \quad & k_{s,e_M,c+1} + k_{s,e_H,c+1} = k_{s,e_M,c} \end{aligned}\right\} \quad \begin{aligned} s &= 1 \ldots S-1, \\ c &= 0 \ldots C-2 \end{aligned}$$

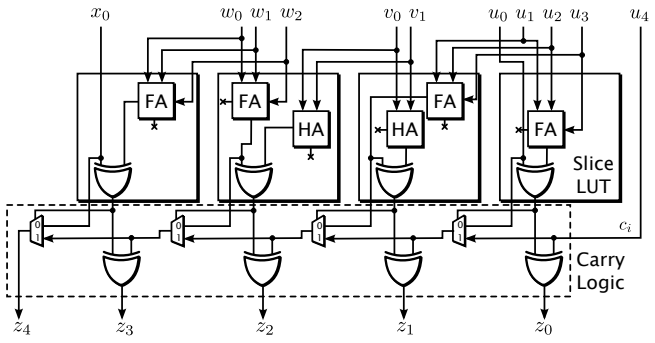
The first constraint (C5) guarantees that if there are compressors of type e_L in column c, there must be the same number of e_M compressors in the succeeding column $c+1$. A similar way was used for constraint C6, namely that if there are e_M compressors in column c, an identical number of e_M and e_L compressors have to be in column $c+1$.

V. Proposed GPCs for Xilinx FPGAs

Although the ILP optimization yields in optimal solutions, it strongly depends on the quality of the set of available compressors. It was shown previously that compressors with high efficiency can be obtained by exploring the low-level architecture of the FPGA [13]. More compressors can be obtained from that idea which are presented in this section. The basic idea is to take an FPGA slice and to push as much compression logic as possible into the LUTs while using the



(a) $(6, 0, 6; 5)$ GPC



(b) $(1, 3, 2, 5; 5)$ GPC

Fig. 5: Proposed GPC slice mappings for Xilinx FPGAs

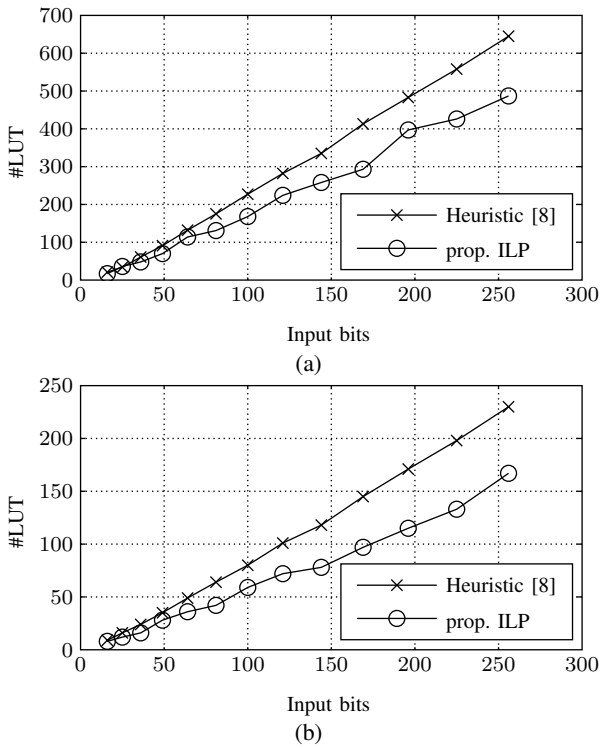


Fig. 6: Resulting number of LUTs over the number of input bits using the heuristic [11] and the proposed ILP model for (a) Virtex 4 and (b) Virtex 6 FPGAs

carry-chain. A slice mapping of a novel $(6, 0, 6; 5)$ GPC is shown in Fig. 5(a). In the least significant LUT (the rightmost one), two full-adders are used to compute the sum out of five inputs. An additional input of the same column is added by the full-adder of the carry chain. The shown XOR gates are necessary to complete the carry logic to a ripple carry adder (RCA). A similar structure is used in the second LUT, but now the two carry bits are computed and fed to the RCA. This structure is repeated, leading to the $(6, 0, 6; 5)$ GPC. Its efficiency is $E = 1.75$ which is the highest efficiency reported so far. Even the ternary adder or the $4 : 2$ compressor have a lower efficiency of $E = 1.5$ for the same size ($k = 4$). Its critical path only consists of a single LUT delay plus four stages of fast carry propagation. A GPC with different input configuration is shown in Fig. 5(b), namely the $(1, 3, 2, 5; 5)$ GPC. Although it has a lower efficiency of $E = 1.5$, it may be favorable in cases where not all of the inputs of the $(6, 0, 6; 5)$ GPC are utilized. The delay is identical to that of the $(6, 0, 6; 5)$ GPC. Note that the carry-in of the $(6, 0, 6; 5)$ GPC can not be used as additional input due to routing constraints within the slice (when the 0-input of the carry-chain MUX is fed from a slice input).

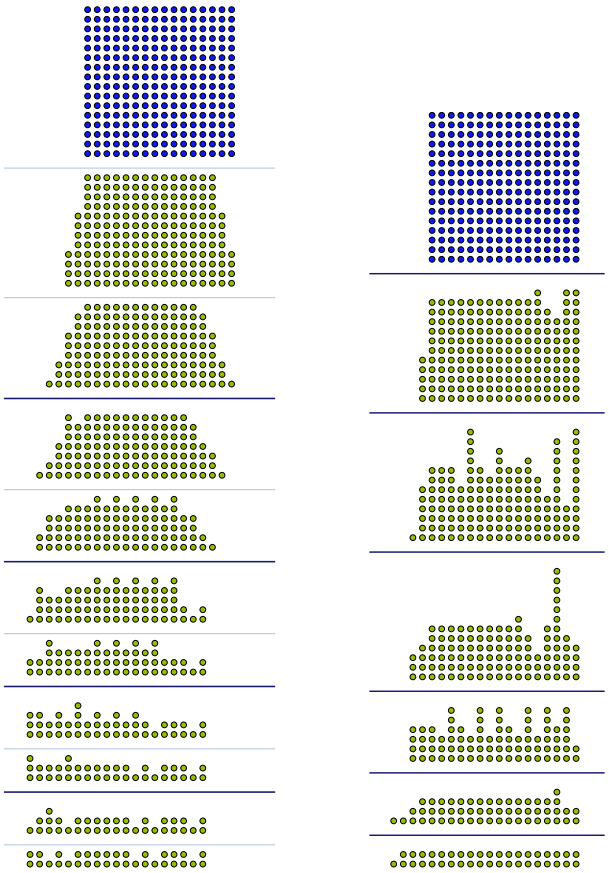
VI. Results

The proposed ILP formulation was integrated within the open-source arithmetic core generator FloPoCo[1] [17], which is a nice framework that supports the handling of compressor trees as a bit heap (including signed number support) [11] as well as the support for VHDL code generation and automated tests. It also includes the recently proposed heuristic compression method [11] which makes it perfectly suitable for comparisons as both methods work on identical data structures and use the same VHDL generation framework. To be able to provide our method as open-source tool it was decided to use the open-source ILP solver SCIP [16], although it is well known that the commercial CPLEX ILP optimizer is much faster (we observed speedups about $10\times$ which is confirmed by a benchmark provided at [16]).

A. Evaluation of the Optimization Quality

To evaluate the performance of the compression we implemented a multiple-input adder with a variable number of inputs as well as a variable word size. We chose this type of circuit because it uses only the compressor tree plus an additional VMA at the output. As VMA we chose a common two-input adder for performance reasons. In the experiments we target Virtex 4 and Virtex 6 FPGAs from Xilinx as candidates with different LUT input sizes. For Virtex 4 (4-input LUTs), we used the same LUT-based GPCs that are used in the FloPoCo framework, namely the $(3; 2)$, $(4; 3)$ and $(1, 3; 3)$ GPCs with LUT cost 2, 3 and 3, respectively. FloPoCo allows the specification of a target frequency to decide how many pipeline stages are used. This frequency was set to

[1]Currently managed in a branch of the FloPoCo subversion repository



(a) (b)

Fig. 7: Compression of a 16-bit 16-input adder: (a) FloPoCo Heuristic, (b) Proposed ILP solution

600 MHz for the heuristic to yield similar timing results and thus comparable resource consumptions. The input word size as well as the word length were varied from 4 to 16 leading to rectangular bit heaps of size 16 to 256 bit. As the ILP optimization may be very time-consuming, SCIP was set to a time limit of 1 hour, which we thought is reasonable. In most cases, a valid solution was found within seconds or minutes which already outperformed the heuristic results. The current implementation allows to interrupt the optimization at any point and VHDL code is generated for the best solution found so far (if any).

The resulting LUT cost from the optimization using the heuristic [11] and the proposed ILP method (with respecting flip-flops cost for pipelining) are shown in Fig. 6(a). It can be observed that the LUT costs follow a fairly linear trend related to the number of bits, independent of the method used. However, the proposed method has a much lower gradient of 1.9 LUT/bit compared to 2.5 LUT/bit. The average LUT reduction is 22.8%. Up to a complexity of 100 bits, an optimal solution was always found within the given time limit, often within a few seconds. As the trend in Fig. 6 continues for higher complexities, it can be assumed that the non-optimal solutions are not too far from being optimal.

TABLE III: Synthesis results for Virtex 4 FPGA

	Heuristic [11]				proposed ILP			
Size [bits]	LUT4	FF	Slices	f_{max} [MHz]	LUT4	FF	Slices	f_{max} [MHz]
16	34	20	25	501.5	28	21	25	562.4
25	45	39	29	455.2	46	45	39	562.1
36	78	63	59	489.5	54	56	35	491.4
49	123	86	73	444.8	79	78	46	481.9
64	181	108	109	412.9	123	120	100	471.5
81	209	132	117	420.7	141	135	106	477.8
100	267	173	174	414.8	181	178	109	454.6
121	332	182	181	332.6	242	247	211	435.4
144	395	243	255	376.2	272	273	223	441.1
169	492	283	277	344.8	309	317	197	428.3
196	582	328	368	355.0	407	416	340	423.2
225	622	345	410	333.9	444	451	349	424.3
256	706	386	459	343.3	506	518	438	410.3
Avg.:	312.8	183.7	195.1	401.9	217.8	219.6	170.6	466.5
Imp.:	–	–	–	–	30.3%	-19.6%	**12.5%**	**16.1%**

The same procedure was applied to Virtex 6 FPGAs which allow the use of 6-inputs LUTs. Here, much more LUT-based GPCs are possible and some of them use the fact that 6-input LUTs can be configured to two 5-input LUTs with shared inputs (which is the case for GPCs with five or less inputs). The LUT-based GPCs used from the FloPoCo framework have the configurations $(6;3)$, $(1,5;3)$, $(5;3)$, $(1,4;3)$, $(4;3)$, $(2,3;3)$, $(1,3;3)$. In addition to that, we used the fastest of the Virtex 6 optimized GPCs from [13] $(1,4,1,5;5)$, $(1,4,0,6;5)$ and $(2,0,4,5;5)$ as well as the $(1,3,2,5;5)$ and $(6,0,6;5)$ GPCs proposed above for the ILP optimization. The results are shown in Fig. 6(b) which shows a similar progression. The obtained gradient is 0.65 LUT/bit compared to the 0.9 LUT/bit of the heuristic. The average LUT reduction is 30.4%.

The number of compressor stages could be drastically reduced compared to the heuristic, even though the optimization goal was the cost minimization (the stage minimization of Section IV-C was not used). The average reduction of stages is 45.1% (from 7.8 stages to 4.3) and 28.3% (from 3.5 stages to 2.5) for Virtex 4 and Virtex 6, respectively. The compression of the largest 256 bit instance is shown in Fig. 7. While the heuristic tries to reduce the maximum height in each step (Fig. 7(a)) there are some pretty high columns in the ILP solution (Fig. 7(b)). It is interesting to note that this is contrary to the well known compression using full-adders. There, the strategy for minimal depth is to always first reduce the largest column [2]. Hence, one can conclude that the reduction of the highest column may not always be the best strategy when using GPCs.

B. Synthesis Experiments

All designs from the previous subsection were also synthesized for Virtex 4 (XC4VLX100-10FF1148) and Virtex 6 (XC6VLX760-FF1760) FPGAs using Xilinx ISE v13.4. The standard ISE settings often result in designs with poor slice utilization, i. e., full slices are often used to implement single

TABLE IV: Synthesis results for Virtex 6 FPGA

	Heuristic [11]				proposed ILP			
Size [bits]	LUT6	FF	Slices	f_{max} [MHz]	LUT6	FF	Slices	f_{max} [MHz]
16	12	7	3	478.0	10	9	3	639.4
25	24	11	6	636.5	26	25	7	452.9
36	32	13	9	595.6	27	36	7	603.1
49	44	15	12	492.4	35	40	10	407.7
64	59	19	16	407.7	47	48	13	506.8
81	76	21	20	442.9	56	59	15	480.1
100	96	47	26	435.9	77	98	20	437.5
121	116	26	32	401.6	89	112	25	438.6
144	134	28	35	383.9	94	121	24	469.0
169	161	60	43	396.8	119	155	30	470.6
196	189	76	50	358.0	131	160	35	408.0
225	216	81	56	327.2	192	236	57	364.0
256	251	74	66	338.3	204	251	55	372.3
Avg.:	108.5	36.8	28.8	438.1	85.2	103.8	23.2	465.4
Imp.:	–	–	–	–	21.5%	-182.4%	**19.5%**	**6.2%**

LUTs. Hence, the slice utilization was obtained by setting the map tool to a minimum *packfactor* (option `-c 1`) yielding a maximum packing density. This leads to more realistic results as the situation is similar to a device utilization close to 100%. Even if the routing may not possible in such situations, a valid routing was found in all the considered cases. To get a realistic timing evaluation, the maximum frequency was obtained by inserting registers at the inputs. The synthesis results for the LUT, FF and slice usage as well as the maximum clock frequency are listed in Table III and Table IV. Even though the number of flip-flops is significantly higher in the proposed method, the slices could be reduced by 12.5% and 19.5% on average for Virtex 4 and Virtex 6, respectively, while achieving a higher performance. This results from the fact that most of the flip-flops are located in the same slice as the compressors do and, thereby, not require extra resources.

VII. Conclusion

A novel method for optimizing compressor trees on FPGAs based on ILP is proposed. In contrast to previous ILP formulations [4], [6], [8] it focusses on minimizing the resources for high throughput pipelined designs rather than minimizing the combinatorial delay. It was shown that average LUT reductions of 22.8% for 4-input FPGAs and 30.4% for modern 6-input LUT FPGAs (using novel GPCs in addition) compared to a recent heuristic [11] can be achieved. This leads to average slice reductions of 19.5% for Virtex 6 FPGAs. The ILP formulation is flexible and can be simply adjusted for a minimal stage count and compressors with variable columns.

Further work has to be done for comparison with previous work [4], [6], [8] and to reduce the complexity in case an optimal solution can not be found in a reasonable time. Instead of optimizing the whole compressor tree it could be simply applied stage-by-stage. Alternatively, a complexity reduction could be achieved by a preprocessing stage, where large high-efficient compressors (like the $4:2$ compressor or the $(6,0,6;5)$ GPC) are applied to the bits they perfectly cover. This should drastically reduce the problem size and should still be close to the optimum.

References

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

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

[3] 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.

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

[5] 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, 2011.

[6] T. Matsunaga, S. Kimura, and Y. Matsunaga, "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.

[7] ——, "Multi-Operand Adder Synthesis Targeting FPGAs," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E94-A, no. 12, pp. 2579–2586, Dec. 2011.

[8] ——, "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.

[9] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, "A Compact High-Speed Parallel Multiplication Scheme," *IEEE Transactions of Computers*, no. 10, pp. 948–957, 1977.

[10] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Elsevier, 2004.

[11] 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.

[12] 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.

[13] 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.

[14] ——, "High Speed Low Complexity FPGA-Based FIR Filters Using Pipelined Adder Graphs," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2011, pp. 1–4.

[15] IBM. (2014, Feb.) High-Performance Mathematical Programming Solver for Linear Programming, Mixed Integer Programming, and Quadratic Programming. [Online]. Available: http://www.ibm.com/software/commerce/optimization/cplex-optimizer/

[16] Konrad-Zuse-Zentrum für Informationstechnik Berlin. (2014) A MIP Solver and Constraint Integer Programming Framework. [Online]. Available: http://scip.zib.de

[17] (2013) FloPoCo Project Website. [Online]. Available: http://flopoco.gforge.inria.fr