Improved Synthesis of Compressor Trees on FPGAs by a Hybrid and Systematic Design Approach

Hadi Parandeh-Afshar[1], Arkosnato Neogy[3], Philip Brisk[2] and Paolo Ienne[1]
[1]*Ecole Polytechnique Fédérale de Lausanne (EPFL)*
[2]*University of California Riverside* / [3]*Indian Institute of Technology, Kharagpur*
[1]*{hadi.parandehafshar and paolo.ienne}@epfl.ch* / [2]*philip@cs.ucr.edu* / [3]*a.s.neogy@gmail.com*

Abstract

Improving arithmetic circuits on FPGAs is one of the main imperatives of FPGA vendors. Augmenting logic cells with dedicated arithmetic components such as adders and carry chains indicates the need for such improvements. In a prior work, we showed how the carry chains in the state-of-the-art Altera FPGAs could be exploited for synthesis of compressor trees. In that work, we proposed generalized parallel counters (GPCs) as the building blocks and mapped them to logic cells of FPGA using LUTs and carry chains. In this paper, we propose a novel technique to increase the logic density of compressor tree synthesis by sharing the logic cells between two neighbor GPCs in a chain. Moreover, we expand the GPC library with bigger GPCs and we propose a systematic approach to select the right GPCs based on the synthesis optimization targets. Finally, we will demonstrate that our framework can be retargeted to Xilinx Virtex-5 FPGAs with minor modifications.

1. Introduction

THIS paper presents a hybrid synthesis approach for compressor trees on FPGAs. A compressor tree is an efficient circuit-level implementation of multi-input integer addition. Multi-input addition is a key component in many signal processing and multimedia applications. Moreover, a recently-published set of transformations has shown how to automatically optimize an arithmetic circuit to maximize the use of compressor trees [1]. These transformations merge the partial product reduction trees of parallel multipliers with one another, and with disparate addition operations. Consequently, there is a need for improving the synthesis of compressor trees on FPGAs.

In a previous paper [2], we proposed a hybrid top-down and bottom-up approach for the synthesis of compressor trees on Altera Stratix III FPGAs. In this paper, we expand that work in several aspects. First, we propose a novel technique to share the logic cells between two adjacent GPCs, which reduces area and makes synthesis more compact. Second, we have expanded the number of GPCs available as building blocks, including the use of larger (8-input) GPCs. We propose a systematic approach to choose the best GPC for use during each step of the synthesis process based on the optimization goal. Unfortunately, this entire approach cannot be architecture-agnostic due to disparate architectural features among the FPGAs sold by leading vendors in the field. This paper shows that our approach is amenable to both *Altera Stratix III* and *Xilinx Virtex-5* FPGAs.

2. Related work

Poldre and Tammemae [3] developed a similar mapping technique to ours for the carry chains in the original *Xilinx Virtex* FPGA; their mapper effectively synthesized soft multipliers on the LUTs, using the carry chains. Their approach is not amenable to the carry chains used in more recent high-performance FPGAs.

This work is an extension of previous papers by Parandeh-Afshar et al. [2, 4] that have studied heuristic methods to synthesize compressor trees on FPGAs. The first paper synthesized GPCs with 6-inputs and 3- or 4-outputs onto 6-LUTs on *Stratix II* FPGAs, and did not use carry chains [4]. This approach was inefficient, as only one 6-LUT in an ALM could be used, due to the restriction that both 6-LUTs must implement the same logic function. To address these concerns, a method to synthesize GPCs using fracturable LUTs and carry chains in *Arithmetic Mode* was introduced [2]. This significantly improved the area and led to improvements in delay.

This paper extends these two prior papers in several respects. Firstly, we consider GPCs with more than 6 inputs, operating on the granularity of LABs rather than ALMs. Secondly, we consider the *Xilinx Virtex* family of FPGAs for the first time, specifically, the *Virtex-5*.

Lastly, we reduce the area by sharing the logic cells between GPCs.

Stenzel et al. [5] introduced the use of generalized parallel counters (GPCs) to realize compressor trees; our synthesis method also uses GPCs. Our approach, however, is quite different in that we tailor the GPCs to match the specific features of the target FPGA.

Matsunaga et al. [6] also developed optimal integer linear programming methods to build compressor trees from GPCs. This paper does not employ carry chains.

3. Arithmetic preliminaries

Both *Altera* and *Xilinx* FPGAs have the variations of the ripple-carry adder. In such adder there is carry propagation and the delay of an *n*-bit ripple carry adder is *O(n)*. The building blocks of a ripple carry adder is the *Full Adder (FA)*.

Let $A_1, \ldots, A_k$, $k > 2$, be a set of integers. A *compressor tree* takes $A_1, \ldots, A_k$ as inputs and outputs two values, *S (sum)* and *C (carry)* where $S+C = A_1 + \ldots + A_k$. A two-input adder is then required to compute $S+C$. Unlike a carry-propagate adder, there is no carry propagation between the same level building blocks of the compressor tree.

A *parallel counter* is a circuit that takes *m* inputs and produces $n = \lceil \log_2(m+1) \rceil$ outputs, such that the outputs are the unsigned integer representation of the number of input bits that are set to '1'.

Each bit b_i in position i of an unsigned integer contributes a total of $b_i 2^i$ to the total value of the integer; the position, i, is called the *rank* of the bit. A *generalized parallel counter (GPC)* is a parallel counter that can takes input bits of varying ranks. A GPC is represented as a tuple $(k_n, k_{n-1}, \ldots, k_0; n)$, where k_i denotes the number of input bits of rank i, and n is the number of output bits. The largest output value that a GPC can produce is $M = k_0 2^0 + \ldots + k_{n-1} 2^{n-1} + k_n 2^n$, so $n = \lceil \log_2(M+1) \rceil$. A parallel counter, as described in the preceding section, is a degenerate GPC. Our compressor tree synthesis method uses GPCs as the building blocks. GPCs map quite efficiently onto FPGA logic cells, and have greater flexibility than parallel counters alone, especially when the input bit pattern is irregular.

4. FPGA Architecture

This section summarizes the key features of the FPGAs that we target: the *Altera Stratix III* and the *Xilinx Virtex-5*. These FPGAs are fabricated in 0.65nm CMOS technology. Both feature 6-LUTs and carry chains, and offer hardware support for 3-input addition; however, there are numerous differences between them as well.

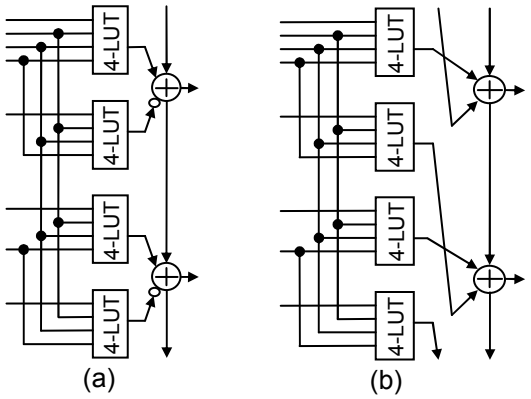

Fig. 1. The Stratix II-IV ALM shown in arithmetic mode (a) and shared arithmetic mode (b).

4.1. Altera-Stratix-III

The *Adaptive Logic Module (ALM)* is the logic cell used by *Altera* in the *Stratix II-IV* series of FPGA [3]. Each ALM contains a variety of *look-up table (LUT)*-based resources that can be divided between two *adaptive LUTs (ALUTs)*. The two 6-LUTs in an ALUT share four common inputs. Moreover, the LUTs are fracturable, meaning that the 6-LUTs are composed of smaller LUTs that are themselves addressable.

An ALM can operate in one of the following modes: *Normal*, *Extended LUT*, *Arithmetic* and *Shared-Arithmetic*. *Normal* and *Extended LUT* mode use the fracturable LUTs exclusively, and do not make use of the carry chain. *Arithmetic Mode*, depicted in Fig. 1(a), decomposes each 6-LUT into a pair of 4-LUTs with shared inputs, each of which can implement a small logic function, whose outputs drive a ripple-carry adder; some input signals are shared between the LUTs.

Shared Arithmetic Mode, depicted in Fig. 1(b), is similar to *Arithmetic* mode, but changes the interconnection pattern between the 4-LUTs and the ripple-carry adder. The 4-LUTs can be configured as a carry-save adder, whose outputs drive the ripple carry adder; this provides an efficient implementation of a 3-input adder.

4.2. Xilinx-Virtex-5

The logic cell of *Virtex-5* is called a *SLICE*. Every slice contains four 6-LUTs, four *xor* gates, and some additional carry logic, including multiplexors. Each LUT implements one arbitrarily-defined 6-input logic function, and LUTs do not share inputs. The 6-LUTs are also fracturable: each 6-LUT can be decomposed into

two 5-LUTs, each of which can be configured to produce a separate logic function.

Signals produced by a LUT can exit the slice, or they can drive the *xor* gate, enter the carry chain, enter the select line of the carry-logic multiplexor, or feed the D input of the storage element. The carry chains and *xor* gates perform fast arithmetic addition and subtraction in a slice.

Fig. 2 shows the structure of a 3-input adder synthesized on a *Virtex-5*. As shown in Fig. 2(a), each LUT is configured as a full adder, which produces a sum and a carry bit; however, the LUT is also configured as part of a second full adder in conjunction with the *xor* gate, multiplexor, and carry chain, as shown in Fig. 2(b); unlike a typical full adder, a multiplexor, which is part of the carry chain, computes the carry-out bit.

5. Proposed synthesis method

This section describes our method for synthesizing compressor trees on FPGAs, while exploiting the architectural features of the logic cells contained in modern high-performance FPGAs. The proposed method has two steps, as shown in Fig. 3.

The first step designs and implements a GPC component library that is tailored to the architectural feature of the target FPGA's logic cells—ALMs/LABs

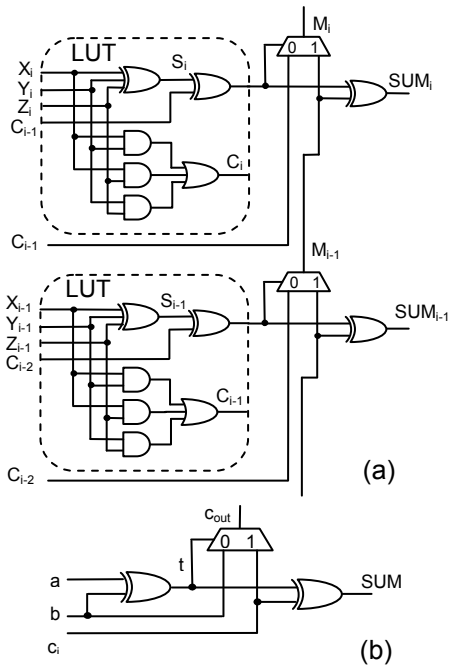

Fig. 2. A Virtex-5 slice configured as a 3-input adder; the functionality implemented by a LUT is circled with a dashed line (a); a full adder designed using a multiplexor to compute the carry-out bit (b).

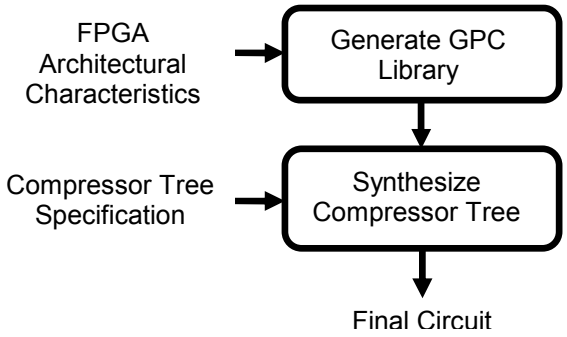

Fig. 3. Overview of how we map compressor trees to FPGAs. First, we manually design a library of GPCs for a specific FPGA architecture. Second, we use the libraries to synthesize a specific compressor tree.

in the case of the *Stratix III* and Slices in the case of the *Virtex-5*. Two implementations, one optimized for delay and one optimized for area, are proposed. Since current FPGA logic cells contain a mixture of LUTs and other dedicated logic gates, different implementations of the same GPC are possible. FPGA CAD tools, most often, only use LUTs to implement GPCs. As a consequence, the GPC library must be implemented manually.

The second step is a heuristic that synthesizes a compressor tree using the library of GPCs generated by the first step. The primary design objective can be to optimize the compressor tree for either latency or area; the objective determines whether the GPCs that are optimized for delay or for area are used. This part of the design flow is completely automated and, once the target library has been specified, is not specific to any particular FPGA.

Subsections 5.1 and 5.2 describe our techniques to generate GPC libraries for the *Stratix III* and *Virtex-5* FPGAs; Subsection 5.3 then presents our compressor tree synthesis heuristic using these different libraries.

5.1 Stratix III GPC Library

As mentioned in Section 3.1, *Stratix III* ALMs have four modes of configuration. In *Normal Mode*, each ALM can implement one 6-input logic function; therefore, any GPC with 6 or fewer inputs and *k* outputs can be implemented in one logic level with *k* ALMs. Increasing the number of GPC inputs beyond 6 will increase the number of ALMs per level, increasing both logic and routing delay.

Using *Arithmetic* and *Shared Arithmetic Mode*, we can implement GPCs with up to 8 inputs by using smaller LUTs (due to the *Stratix III*'s fracturable LUT structure [3]) in conjunction with carry chains. As an example, Fig. 4 shows the circuit and ALM-based implementations of a *(0, 6; 3)* GPC in arithmetic mode. To propagate the input a_0 into the second full adder, it is

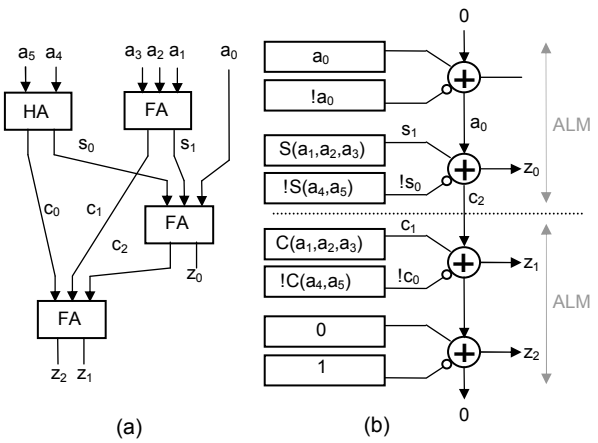

Fig. 4. A (0, 6; 3) GPC implemented at the circuit level (a) and synthesized on ALMs and carry chains using arithmetic mode

necessary to program the first pair of LUTs to implement the functions a_0 and a_0', while putting a value of '0' on the carry-in. As the third input of the full adder is complemented, the full adder effectively computes *FA (0, a_0, a_0)*. The sum output of the full adder is always '0', in this case, and the carry output is a_0. Thus, to efficiently map this GPC onto the ALM, we have configured two LUTs and a full adder in the carry chain as identity functions that propagate a single bit into the carry-in of a full adder further down the chain. It is also important to note that both the carry-in and carry-out values of the GPC on the carry chain are '0'; consequently, several of these GPCs can be abutted to adjacent ALMs in a LAB. These '0' values effectively "break" the logical, carry chain, permitting a new logical chain to start at the next ALM.

This type of GPC design, which uses a combination of LUTs and carry chains to bypass the routing network, is applicable to GPCs with up to 8 inputs. GPCs with more than 8 inputs require more levels of ALMs. Table I shows a list of GPCs that we have synthesized by hand on the *Stratix III*. Two implementations are presented for each GPC: one that uses only 6-LUTs in *Normal Mode*, and another that uses the smaller LUTs in conjunction with carry chains in *Arithmetic* or *Shared Arithmetic Mode*, e.g., Fig. 4.

From Table I, we observe several trends. Firstly, GPCs built using *Arithmetic* or *Shared Arithmetic Mode* always require fewer ALMs than the implementation using *Normal Mode*. Secondly, for GPCs with *6* or fewer inputs, the implementation using *Normal Mode* has a lower delay than the implementation using *Arithmetic or Shared Arithmetic Mode*; this is because the delay only includes LUTs, and does not include the carry chain. Thirdly, for GPCs with 7 or 8 inputs, two layers of ALMs are required for *Normal Mode*; thus, the

Table I. A library of GPCs built using Normal Mode (6-LUTs only) or either Arithmetic or Shared Arithmetic Mode (4-LUTs + carry chains).

GPC	Normal Mode		(Shared) Arithmetic Mode	
	Delay (ns)	Area (ALMs)	Delay (ns)	Area (ALMs)
(0,6;3)	0.38	3	0.97	2
(1,5;3)	0.38	3	0.97	2
(2,3;3)	0.38	3	0.97	2
(0,7;3)	1.36	4	0.98	2.5
(2,6;4)	1.36	5	1.01	3
(3,5;4)	1.36	5	1.01	3
(4,4;4)	1.36	5	1.01	3
(5,3;4)	1.36	5	1.01	3
(6,2;4)	1.36	5	1.01	3

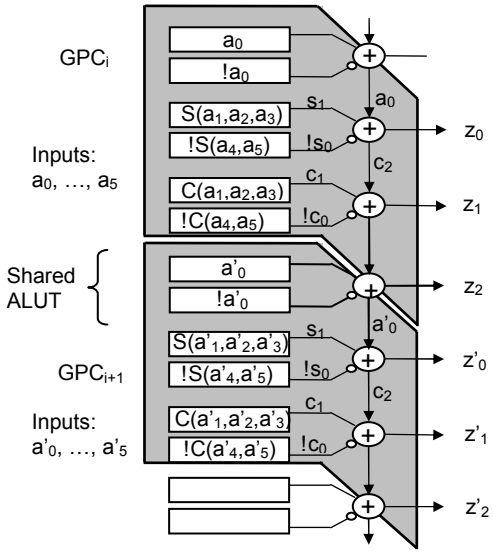

Fig. 5. Two abutted GPCs implemented using the same mode (Arithmetic or Shared Arithmetic) can share an ALUT, improving area utilization.

delay includes the cost of two LUTs, plus routing; for these GPCs, the implementations using *Arithmetic* or *Shared Arithmetic Mode* are faster and use fewer ALMs.

Two GPCs that use the same configuration mode, i.e., either both use *Arithmetic* or *Shared Arithmetic Mode*, can share some ALMs when they are abutted. Looking back at Fig. 4, this is possible because the first ALUTs in a GPC do not produce any output, and the last ALUT does not require any inputs. Therefore, it is possible to share the last ALUT of one GPC with the first ALUT of the next. Fig. 5 shows an example, in which two of the *(0, 6; 3)* GPCs from Fig. 4 are abutted together, requiring 3 ALMs rather than 4. In this figure, the fourth ALUT is shared between the two GPCs.

Referring back to Table I, the 6-input GPCs all require *Arithmetic Mode*; the GPCs with more than 6 inputs require *Shared Arithmetic Mode*. With respect to the first group, *n* GPCs can be abutted requiring *3n+1*

Table II. A library of GPCs synthesized on the Virtex-5 using 6-LUTs only, or a combination of LUTs with carry chains.

GPC	LUTs only		LUTs + Carry Chains	
	Delay (ns)	Area (SliceLUTs)	Delay (ns)	Area (SliceLUTs)
(0,6;3)	0.35	3	1.04	4
(1,5;3)	0.35	3	0.79	3
(2,3;3)	0.35	3	0.79	3
(0,7;3)	1.48	6	1.04	4
(2,6;4)	0.84	7	1.04	4
(3,5;4)	0.65	7	1.04	4
(4,4;4)	0.91	6	1.04	4
(5,3;4)	0.65	5	1.04	4
(6,2;4)	0.91	7	1.04	4

ALUTs. As each ALM contains two ALUTs, and we abut groups of GPCs that use up to five contiguous ALUTs; therefore, we must choose a value of n that satisfies $3n + 1 \leq 10$. Therefore, we can abut up to three GPCs from the first group with shared LUTs in half of a LAB.

With respect to the second group, n GPCs can be abutted requiring $4n+2$ ALUTs; thus, we must choose a value of n that satisfies $4n + 2 \leq 10$. Therefore, we can abut at most two GPCs from the second group with shared LUTs in half of a LAB. Without sharing LUTs, we could only pack one of these GPCs in the same space.

5.2 Virtex-5 GPC Library

Slices and CLBs in the *Virtex-5* FPGA are significantly different from the ALMs in the *Stratix III*; thus, the GPC library discussed in the preceding section, and the LUT sharing strategy, cannot be used. In section 3.2, we showed how a 3-input adder can be implemented in the *Virtex-5*. As explained, the 6-LUT

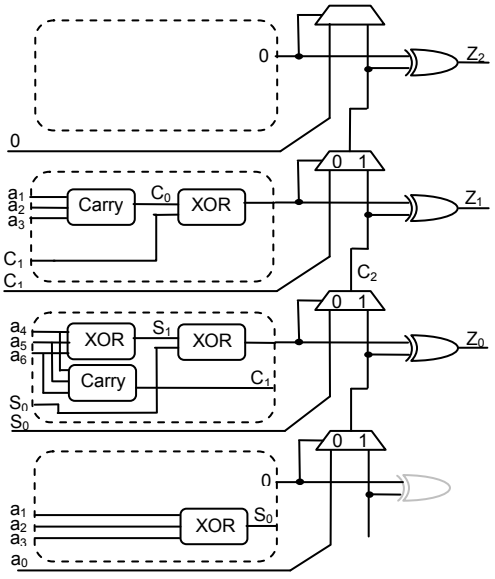

Fig. 6. A (0, 7; 3) GPC synthesized on two Virtex-5 Slices.

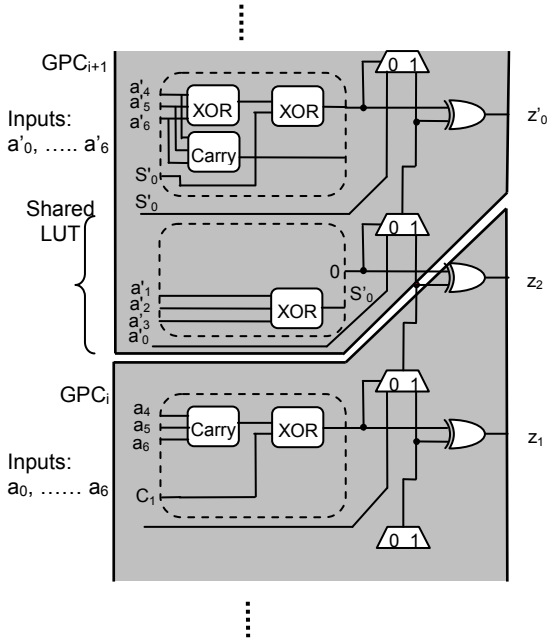

Fig. 7. Two (0, 7; 3) GPCs abutted using shared LUTs to reduce area. Only portions of both GPCs are shown to conserve space.

can be decomposed into two 5-LUTs which produce two outputs. For 3-input adders, one of the 5-LUT inputs should be the carry output of the preceding bit position, which leaves only four free inputs remaining.

Fig. 6 shows a *(0, 7; 3)* GPC synthesized on the Virtex-5; the circuit-level diagram of this GPC is the same as the one shown in Fig. 4(a). The carry output of the second LUT does not depend on the output bit of the first LUT; this prevents the formation of a multi-LUT critical path involving carry chains. Similar to the *Stratix III*, we can also build GPCs with up to 8 inputs on the *Virtex-5*. 6-input GPCs achieve better delay and area utilization when using LUTs only. Table II shows the resulting GPCs.

Similar to the *Stratix III*, the resources could be shared for two adjacent GPCs as shown in Fig. 7. This figure shows an example for *(0, 7; 3)* GPCs; in this case, the last LUT of the first GPC and the first LUT of the second GPC can be shared.

5.3 Compressor Tree Synthesis Heuristic

This section presents an architecturally-agnostic synthesis heuristic for compressor trees that uses the target-specific libraries described in the preceding subsections. First, we characterize each GPC in the library in terms of its ability to reduce the number of bits, its delay, and its area.

The *compression difference (CD)* of a GPC is the difference between the number of inputs to outputs; for example, the compression difference of a *(2, 3; 3)* GPC is *5 – 3 =2*.

We must consider the delay of each GPC as well. GPCs with larger compression differences also have larger delays, so the choice of when to use a given GPC is non-obvious. The *Performance Degree (PD)* of a GPC is defined as the ratio *PD = CD / Delay*, which favors GPCs with higher compression differences and smaller delays.

To reduce the area required for a compressor tree, we introduce the *Area Degree (AD)*, defined to be the ratio *AD = CD / Area*. The area degree favors GPCs with higher compression differences and smaller areas.

Lastly, the *Area-Performance Degree (APD)* is *AD×PD*. This metric is used when one wants to simultaneously optimize both delay and area.

We envision three distinct design scenarios with the following objectives:

1. The primary objective is to minimize combinational delay; the area cost is immaterial.
2. The primary objective is to minimize the area-delay product.
3. The primary objective is to minimize area; the circuit can be pipelined to achieve a desired clock frequency.

For the first scenario, we try to maximize the performance degree; for the second scenario, we try to maximize the area-performance degree; and for the third, we try to maximize the area degree.

The mapping heuristic is the same for all three objectives; the three objective criteria outlined above are used to sort the GPCs in a priority order. At each step, the heuristic traverses the prioritized list of GPCs, and selects the first one that it can use in the situation. Tables III and IV show the priorities assigned to each GPC listed in Tables I and II respectively, for each of the three objectives. The values for *AD* and *APD* assume that LUT sharing is used whenever possible.

Fig. 8 shows pseudo code for the compressor tree synthesis heuristic. There are four parameters: an array of integers, *columns*, two integers, *M*, and *N*, and a value, *mode*, which is one of *{PD, AD, APD}*. A column refers to the number of bits in each input integer having the same rank, e.g., column *i* contains all input bits of rank *i*. *M* is the maximum number of GPC inputs supported by the heuristic, and *N* is the maximum number of outputs supported (*M = 8* and *N = 4* for the GPC libraries summarized in Tables I-IV). The output is a compressor tree built as a network of GPCs.

The Mapping heuristic has four main steps.

Step 1: In the first step, given *M* and *N*, a library of GPCs is formed. At present, this step is performed by hand, as current FPGA CAD tool cannot achieve the best possible mappings of GPCs onto FPGA resources for either the *Stratix III* or *Virtex-5*. The user also supplies the parameter *mode* so that the library construction algorithm can sort the GPCs in non-decreasing order of *PD*, *AD*, or *ADP*.

In each of these scenarios a different GPC has the highest priority. Considering the *Stratix III*, for Scenario 1 (maximize *PD*), two GPCs share the maximum *PD* value: *(0, 6; 3)* and *(1, 5; 3)*, both implemented using *Normal Mode*; the former is chosen as the highest-priority GPC. For Scenarios 2 and 3 (maximize *APD* and *AD* respectively), the *(0, 7; 3)* GPC implemented using *Shared Arithmetic Mode* has the highest priority, as it has the highest *APD* and *AD* values.

For the *Virtex-5*, the highest priority GPC under Scenarios 1 and 2 is *(0, 6; 3)*, implemented using LUTs only; for Scenario 3, the highest priority GPC is *(0, 7; 3)* implemented using LUTs and carry chains.

Lastly, as explained, in all cases a single column GPC has the highest priority, and we represent such a GPC as a *(0, BH; $\lceil log_2(BH+1) \rceil$)*, where BH denotes the column height of the GPC.

Step2: In this step, we process columns one-by-one. Assuming that *column[i]* $\geq$ *BH*, we repeatedly remove *BH* bits from *column[i]*; each group of bits removed are assigned as inputs to a new instance of a maximum-

Table III. The PD, AD, and APD values for the GPCs in Table 1 (Stratix III).

	Normal Mode				Arithmetic and Shared Arithmetic Mode		
GPC	CD	PD	AD	APD	PD	AD	APD
(0,6;3)	3	7.9	0.9	7.1	3.1	1.8	5.6
(1,5;3)	3	7.9	0.9	7.1	3.1	1.8	5.6
(2,3;3)	2	5.3	0.6	3.2	2.1	1.2	2.5
(0,7;3)	4	2.9	0.8	2.3	4.1	2.9	8.2
(2,6;4)	4	2.9	0.8	2.3	4.0	1.6	6.3
(3,5;4)	4	2.9	0.8	2.3	4.0	1.6	6.3
(4,4;4)	4	2.9	0.8	2.3	4.0	1.6	6.3
(5,3;4)	4	2.9	0.8	2.3	4.0	1.6	6.3
(6,2;4)	4	2.9	0.8	2.3	4.0	1.6	6.3

Table IV. The PD, AD, and APD values for the GPCs in Table 2 (Virtex-5).

	LUTs Only				LUTs + Carry Chains		
GPC	CD	PD	AD	APD	PD	AD	APD
(0,6;3)	3	8.5	1.0	8.5	2.9	0.6	1.7
(1,5;3)	3	8.5	1.0	8.5	3.8	1.0	3.7
(2,3;3)	2	5.5	0.6	3.3	2.5	0.6	1.5
(0,7;3)	4	1.2	0.6	0.7	3.8	1.3	4.8
(2,6;4)	4	4.8	0.5	2.3	3.8	1.0	3.8
(3,5;4)	4	6.2	0.5	3.0	3.8	1.0	3.8
(4,4;4)	4	4.4	0.6	2.6	3.8	1.0	3.8
(5,3;4)	4	6.2	0.8	4.9	3.8	1.0	3.8
(6,2;4)	4	4.4	0.5	2.1	3.8	1.0	3.8

```
Mapping_algorithm(Array of Integers : columns,
                  Integer : M, Integer : N
                  mode : {PD, AD, APD} )
{
Step1:  Build_GPC_library(M, N, mode);
        BH = Num inputs of Max. Priority GPC
        Repeat
        {
            while (col_indx≤max_col_indx)
            {
Step2:       if(columns[col_indx] ≥ BH)
               Map_by_GPC();
             else
               col_indx++;
            }

            lsb_to_msb_covering()
            {
                for(i = 0 to max_col_indx)
Step3:           for(j = 0 to num_of_gpcs)
                    if(match(gpc[j],column[i])
                        bind_gpc[i,j];
            }

            Connect_GPCs_IOs( );
Step4:      Propagate_comb_delay( );
            Generate_next_stage_dots( );
        } until at most 3 bits per column remain;
}
```

Fig. 8. Pseudo-code for the compressor tree synthesis heuristic.

priority GPC. We stop when *column[i]* < *BH*. Thus, we repeatedly compress bits using the highest priority counter until all columns contain fewer than *BH* bits.

Step3: Step 3 maps the remaining bits in all of the columns to other GPCs in the library. The heuristic traverses the individual columns in order from least significant to most significant; each column is progressively covered by a GPC. To bind a GPC to the current column, *column[i]*, two conditions must be met. First, we must find a GPC with exactly *column[i]* bits in its least significant position. Second, the subsequent column(s) should have at least as many bits as the number of bits in that position in the GPC. The GPC list is searched in priority order, and the highest priority GPC that satisfies both conditions is chosen. After a GPC has been chosen, the bits that are bound to it are removed from their respective columns. The process then continues, starting with *column[i+1]*, stopping at the most significant column.

Step4: Steps 2 and 3 generate a single level of the compressor tree. This step connects the current level to the previous level, generates the bits for the next level, and controls the stopping condition of the heuristic. The heuristic terminates when all columns contain three bits or fewer. With 3-bits per column, all remaining bits can be summed using a 3-input adder, which can be synthesized on either the *Stratix III* in *Shared Arithmetic Mode* or the *Virtex-5* using LUTs and carry chains.

6. Experimental Evaluation

To evaluate the proposed synthesis method, we took several the multi-input addition units of several real benchmarks and for each FPGA target we synthesized each benchmark in four different approaches:

3-Add: Using the 3-input adder tree of the FPGAs, serving as the baseline for comparison.

Scen-1: The proposed heuristic (Fig. 8) with GPCs prioritized by performance degree (PD).

Scen-2: The proposed heuristic (Fig. 8) with GPCs prioritized by area-performance degree (APD).

Scen-3: The proposed heuristic (Fig. 8) with GPCs prioritized for area degree (AD).

We modeled, by hand, each of the GPCs in the component libraries summarized in Tables I-IV. For the *Stratix III*, we used atom-level modeling to characterize the GPCs. We used the *Verilog Quartus Module (VQM)* format as provided by the *Quartus-II University Interface Program (QUIP)*. VQM is a restricted subset of the *Verilog* language, in which the basic components are logic cells with user-specified parameters such as LUT masks and the configuration mode. These models were used as components from which larger compressor trees were constructed. Using *Quartus-II*, we synthesized each compressor tree on the *Stratix III*; the delay and area numbers reported here are taken from the *Quartus-II* project reports.

To model atom-level GPCs using the *Virtex-5*, we used the primitive components of logic cells, such as LUTs, multiplexors, and *xor* gates; we used a *Verilog*-like format similar to VQM. We used *Xilinx*'s *ISE 8.2* CAD tools for all experiments targeting the *Virtex-5*.

Fig. 9 shows the critical path delay of each of the four synthesis methods outlined above. In general, Scen-1 and Scen-2 achieve the lowest critical path delays, although there are a few exceptions such as FIR6. 3-ADD suffers from the largest critical path delay for all benchmarks, except for G.721, where Scen-3 has a slightly higher delay.

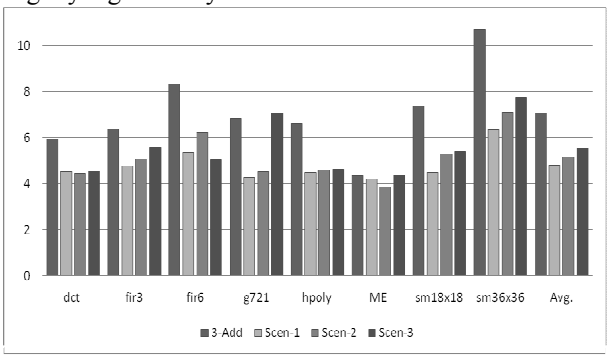

Fig. 9. The critical path delay (ns) of each benchmark synthesized on the Stratix III.

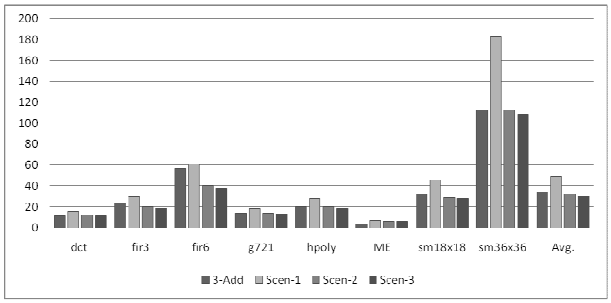

Fig. 10. Area usage (ALMs) of the four synthesis methods on the Stratix III.

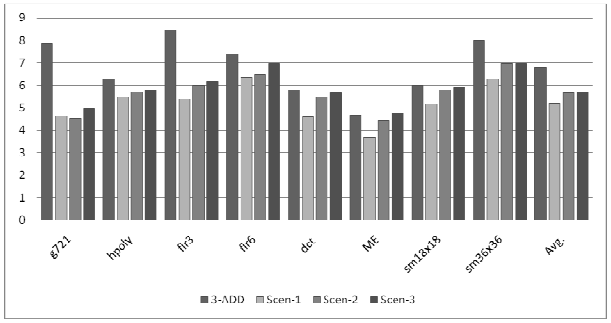

Fig. 11. The critical path delay (ns) of each benchmark synthesized on the Virtex-5.

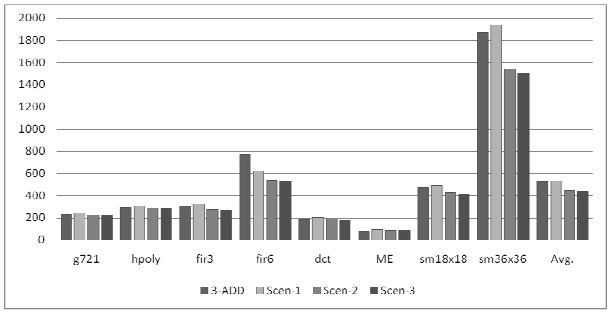

Fig. 12. Area usage (Slice LUTs) of the four synthesis methods on the Virtex-5.

Scen-3, it should be noted, is intended for pipelining. 3-Add is difficult to pipeline, because carry chains do not contain internal registers, whereas, each ALM output has a bypassable register. For this reason, we do not view the large critical path delays of Scen-3 to be a negative result.

Fig. 10 explores area utilization. Scen-1 uniformly requires more LABs than the other synthesis methods, while in most cases, Scen-3 requires the fewest, even fewer than 3-Add. Unlike our prior work [19-21], Scen-3 is the first compressor tree synthesis method that we have found that consistently requires less area than 3-Add. In many cases, Scen-2 requires fewer LABs than 3-Add, although there are a few cases that it requires more. Altogether, Scen-2 offers the best overall balance between critical path reduction and area usage for the *Stratix III*.

Fig. 11 shows the critical path delay of each of the four synthesis methods for each benchmark synthesized on the *Virtex-5*. With one exception (G.721), Scen-1 achieves the smallest critical path delay. Unlike the *Stratix III*, Scen-3 achieves a comparable critical path compared to Scen-2, although there are a few where the critical path delays of 3-Add and Scen-3 are comparable. Overall, Scen-1 and Scen-2 most reliably minimize the critical path delay.

Fig. 12 shows the area usage for the *Virtex-5*. Unlike the *Stratix III*, 3-Add does not retain a noticeable advantage in area utilization compared to Scen-1 and Scen-2. Although Scen-1 tends to use the most Slice LUTs, it uses fewer than 3-Add for several benchmarks such as FIR6; moreover, Scen-2 achieves a significant advantage over 3-Add for FIR6 and SM36x36. On average, and for most benchmarks, Scen-3 achieved the best area usage. For non-pipelined circuits, however, Scen-2 appears to once again offer the best tradeoff between critical path delay and area usage.

7. Conclusion

We have shown that compressor trees can be synthesized on FPGAs that are faster and smaller than 3-input adder trees. Our heuristic method is simple and straightforward to implement, but requires a library of GPCs that have been carefully tailored to the architectural feature of the target FPGA. Our results dispute the conventional wisdom that the presence of carry chains and support for 3-input addition in modern FPGAs makes adder trees superior to compressor trees. Our results clearly indicate that this is not the case.

9. References

[1] A. K. Verma, P. Brisk, and P. Ienne, "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol, 27, no. 10, pp. 1761-1774, Oct. 2008, doi:10.1109/TCAD.2008.2003280.

[2] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting Fast Carry-Chains of FPGAs for Designing Compressor Trees," *Proc. Intl. Conf. Field Programmable Logic and Applications (FPL '09)*, pp. 242-249, Aug.-Sept. 2009.

[3] J. Poldre and K. Tammemae, "Reconfigurable Multiplier for Virtex FPGA Family," *Proc. 9th Intl. Wkshp. Field Programmable Logic and Applications (FPL '99)*, pp. 359-364, Aug.-Sept., 1999.

[4] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient Synthesis of Compressor Trees on FPGA," *Proc. 13th Asia and South Pacific Design Automation Conf. (ASPDAC '08)*, pp. 138-143, Jan. 2008, doi:10.1109/ASPDAC.2008.4483927.

[5] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, "A Compact High-Speed Parallel Multiplication Scheme," *IEEE Trans. Computers*, vol. 26, no. 10, pp. 948-957, October, 1977.

[6] T. Matsunaga, S. Kimura, and Y. Matsunaga, "Multi-Operand Adder Synthesis on FPGAs using Generalized Parallel Counters," *Proc. 18th Wkshp. Logic and Synthesis (IWLS '09)*, June, 2009.