# Compressor Tree Synthesis on Commercial High-Performance FPGAs

HADI PARANDEH-AFSHAR, Ecole Polytechnique Federale de Lausanne (EPFL)
ARKOSNATO NEOGY, Indian Institute of Technology, Kharagpur
PHILIP BRISK, University of California, Riverside
PAOLO IENNE, Ecole Polytechnique Federale de Lausanne (EPFL)

Compressor trees are a class of circuits that generalizes multioperand addition and the partial product reduction trees of parallel multipliers using carry-save arithmetic. Compressor trees naturally occur in many DSP applications, such as FIR filters, and, in the more general case, their use can be maximized through the application of high-level transformations to arithmetically intensive data flow graphs. Due to the presence of carry-chains, it has long been thought that trees of 2- or 3-input carry-propagate adders are more efficient than compressor trees for FPGA synthesis; however, this is not the case. This article presents a heuristic for FPGA synthesis of compressor trees that outperforms adder trees and exploits carry-chains when possible. The experimental results show that, on average, the use of compressor trees can reduce critical path delay by 33% and 45% respectively, compared to adder trees synthesized on the Xilinx Virtex-5 and Altera Stratix III FPGAs.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Field Programamble Gate Array (FPGA), look-up table (LUT), carry chain, compressor tree

**ACM Reference Format:**
Parandeh-Afshar, H., Neogy, A., Brisk, P., and Ienne, P. 2011. Compressor tree synthesis on commercial high-performance FPGAs. ACM Trans. Reconfig. Technol. Syst. 4, 4, Article 39 (December 2011), 19 pages.
DOI = 10.1145/2068716.2068725 http://doi.acm.org/10.1145/2068716.2068725

## 1. INTRODUCTION

*Compressor trees* are efficient implementations of multi-operand fixed-point adders, including, for example, partial product reduction trees of parallel multipliers. Compressor trees use carry-save arithmetic, in contrast to the more straightforward *adder trees*, which are implemented using 2- or 3-input carry-propagate adders. Compressor trees occur naturally in arithmetic circuits such as FIR filters and dot-products, among others. Transformations can also be applied to circuits and data flow graphs to rearrange their operations to favor the use of carry-save arithmetic [Verma et al. 2008]. The superiority of compressor trees over adder trees for ASICs has been known since the 1960s [Wallace 1964; Dadda 1965; Stelling et al. 1998]; however, these compressor tree synthesis methods yield poor results when circuits are synthesized on FPGAs.

---

A. Neogy is currently affiliated with the University of California, Berkeley.
Authors' addresses: H. Parandeh-Afshar (corresponding author), Ecole Polytechnique Federale de Lausanne (EPFL), France; email: hadi.parandehafshar@epfl.ch; A. Neogy, University of California, Berkeley, CA; P. Brisk, University of California, Riverside, CA; P. Ienne, Ecole Polytechnique Federale de Lausanne (EPFL), France.

DOI 10.1145/2068716.2068725 http://doi.acm.org/10.1145/2068716.2068725

This article introduces a heuristic method to synthesize compressor trees on FPGAs, which is capable of exploiting carry-chains in limited and appropriate contexts. This method can be used to reduce the critical path delays of multi-operand adders, and multipliers whose partial product reduction trees have been fused with other adders; fixed-bitwidth multiplication and serialized multiply-accumulate operations can be synthesized on DSP blocks, which already contain embedded fixed-point multipliers.

First, the heuristic builds a library of building blocks called *Generalized Parallel Counters* (GPCs), whose implementations are highly tailored to the structure of the *LookUp Tables* (LUTs) in an FPGA; this library construction phase is vendor- and architecture-specific. Each GPC in the library is characterized in terms of the critical path delay of each output and its area. Second, a greedy heuristic synthesizes the compressor tree in an architecture-agnostic manner, using GPCs from the library as building blocks. In principle, the greedy heuristic could be replaced by a more complicated method that achieves a solution of higher quality at the expense of increased runtime.

Our experiments target the Altera Stratix III and Xilinx Virtex-5 FPGAs. For both FPGAs, we observe significant reductions in critical path delay; for the Virtex-5, area is reduced as well; for the Stratix III, the area depends on how the library components are prioritized. If the components are prioritized to reduce critical path delay, then the area of the compressor tree became larger than that of the adder tree; however, prioritizing for area or area-delay product achieved more conservative reductions in critical path delay, but achieved marginal area improvements compared to the adder trees.

## 2. ARITHMETIC PRELIMINARIES

### 2.1. Compressor Trees

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

$$S + C = A_1 + \cdots + A_k.$$

A 2-input carry-propagate adder then computes the sum $S + C$. The Stratix III and Virtex-5 FPGAs support 3-input adder carry-propagate addition, so the compressor trees designed by our algorithm produce three, rather than two, outputs.

Unlike a carry-propagate adder, a compressor tree has no carry-in to carry-out delay; thus multi-input fixed-point addition scales efficiently as $k$ increases. The compressor tree can also directly compute some of the lower-order output bits of $S + C$, reducing the bitwidth of the final carry-propagate adder. When targeting ASICs, portions of the final adder that sum late-arriving bits can be optimized for area, rather than delay [Oklobdzija and Villeger 1995; Stelling and Oklobdzija 1996]. In our experience, this is ineffective for FPGAs, as routing delays are more difficult to predict at the synthesis level, and the presence of carry-chains ensures that the smallest adders (those that use carry-chains rather than LUTs) tend to be the fastest as well.

### 2.2. (Generalized) Parallel Counters

In a $k$-bit unsigned binary integer $B = b_{k-1}b_{k-2}\ldots b_0$, each bit $b_i$ contributes a value of $b_i 2^i$ to the total value of $B$. The index $i$, which denotes the position of $b_i$, is called the *weight*.

An *m:n parallel counter* has $m$ inputs, $n = \lceil log_2(m+1) \rceil$ outputs, and counts the number of input bits that are "1." In a compressor tree, the input bits of an *m:n* counter have the same weight. Half- and full-adder are 2:2 and 3:2 counters respectively. A *Generalized Parallel Counter (GPC)* takes input bits of varying weights. A GPC is represented as a tuple $(k_w, k_{w-1}, \ldots, k_0; n)$, where $k_i$ denotes the number of input bits of

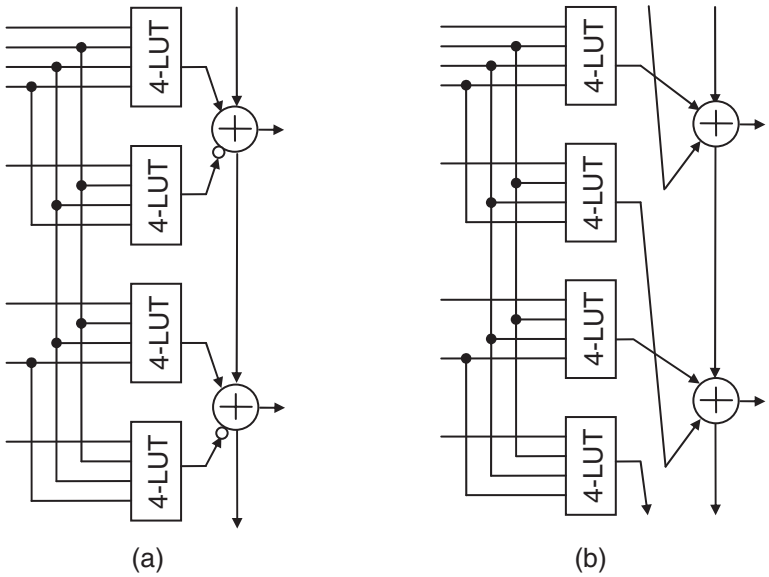


Fig. 1. The Stratix II-IV ALM shown in Arithmetic Mode (a) and Shared-Arithmetic Mode (b).

weight $i$, and $n$ is the number of output bits. The largest output value that a GPC can produce is

$$M = k_0 2^0 + \cdots + k_{w-1} 2^{w-1} + k_w 2^w$$

so $n = \lceil log_2(M+1) \rceil$.

## 3. FPGA ARCHITECTURE AND GPC LIBRARIES

### 3.1. Altera's Adaptive Logic Module (ALM) and Logic Array Block (LAB)

The *Adaptive Logic Module (ALM)* is the logic cell introduced by Altera for the Stratix II FPGA [Altera Corp. 2011]; the Stratix III, IV, and V, which followed, also use the ALM. Each ALM contains a variety of *Look-Up Table* (LUT)-based resources that can be divided between two *Adaptive LUTs* (ALUTs). A variety of LUT sizes are supported by each ALM including two different 5-LUTs and one 6-LUT.

An ALM can operate in one of the following modes: *Normal*, *Extended LUT*, *Arithmetic*, and *Shared-Arithmetic*. Normal and Extended LUT modes use LUTs exclusively and do not make use of the carry-chain. Arithmetic Mode, depicted in Figure 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. Shared-Arithmetic Mode, depicted in Figure 1(b), is similar to Arithmetic mode, but changes the interconnection pattern between the 4-LUTs and the ripple-carry adder. The 4-LUTs are configured as a carry-save adder, whose outputs drive the ripple-carry adder; this provides an efficient implementation of a 3-input adder.

Several ALMs are clustered together with a fast local routing network to form a *Logic Array Block* (LAB). The I/O bandwidth of the LABs is limited, that is, a LAB containing $k$ ALMs, each having 8 inputs, has fewer than $8k$ inputs connected to the adjacent routing network. The number of ALMs per LAB is eight for the Stratix II and ten for the Stratix III, and IV. The carry-chain spans all of the ALMs in a LAB, but a new carry-chain can only start at the first or the sixth ALM in a LAB in the Stratix III and IV.

In Normal Mode, each ALM can implement a 6-input logic function; therefore, any GPC with 6 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 [Hutton et al. 2004]) in conjunction with carry-chains.

### 3.2. The Xilinx Virtex-5 Slice Architecture

The logic cell of Virtex-5 is called a slice [Xilinx Corp. 2011]. Each slice contains four 6-LUTs, four *xor* gates, and additional carry logic, including multiplexers. Each 6-LUT implements one 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. The propagation delay through a LUT is independent of the function that it implements, or whether it implements one 6-input or two 5-input functions. Signals produced by a LUT can exit the slice, drive the *xor* gate, enter the carry-chain, enter the select line of the carry-logic multiplexer, or drive the input of a flip-flop. The carry-chains and *xor* gates perform fast arithmetic addition and subtraction in a slice. Slices are laid out to form columns. Carry-chains can be formed that span all of the slices in a column; that is, the carry-chains are cascadable, permitting them to perform addition or subtraction on operands of arbitrary bitwidth.

A pair of slices forms a *Configurable Logic Block* (CLB). The two slices in a CLB do not connect to one another; each belongs to a different column and has an independent carry-chain. Each CLB connects to a switch matrix for access to the routing network. There is no notion of a LAB-like logic cluster with fast local routing.

Figure 2(a) shows a 3-input adder synthesized on a Virtex-5. Each LUT is configured as a full adder, which produces a sum and a carry bit; this is similar to the carry-save adder synthesized on the Stratix III's ALM using *Shared-Arithmetic Mode*. The second full adder, shown in the shaded box, is formed by the conjunction of the same LUT with the *xor* gate and multiplexer. The *xor* gate's output represents the sum and the multiplexer output represents the carry. The $C_i$ input to each LUT is the output of the previous LUT in the chain, which is connected by a routing wire; however, the design is structured so that $C_i$ is dependent on the inputs to the full adder and not on $C_{i-1}$. For this reason, the path that goes through the routing network is noncritical.

### 3.3. GPC Libraries

GPCs are architecture-specific and are designed manually. Each GPC is implemented twice: using only LUTs, and using a combination of LUTs and carry-chains. Our approach is to model each GPC as a network of full and half-adders. We identify *chains* of full and half- adders within each GPC that are suitably mapped onto carry-chains. The remaining full and half-adders map onto LUTs. Appendix A shows the design of the GPCs in the library that can be mapped to the carry-chains.

Figure 3(a) and 3(b) depict an ALM-based (0, 6; 3) GPC for the Stratix III, respectively, as a network of full and half-adders and after mapping onto the ALM using Arithmetic Mode. The LUT-only implementation of the GPC requires three ALMs: one for each output bit. Similarly, Figures 3(c) and (d) show a (3, 5; 4) GPC implemented by three ALMs using the same approach. The LUT-only implementation of the same GPC requires five ALMs that are connected by routing wires, as the number of GPC inputs exceeds the number of LUT inputs; in contrast, a single layer of ALMs can implement the same GPC by exploiting the carry-chain.

Both adders in Figure 3 need to route input bits directly to the carry-chain, bypassing LUTs. To accomplish this, we exploit the following property of full adders.

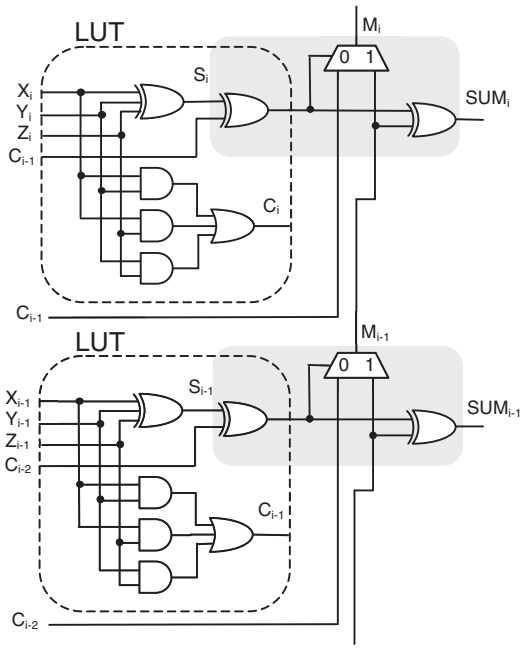


Fig. 2. A Virtex-5 half-slice configured as a 3-input adder; the functionality implemented by a LUT is circled with a dashed line and the shaded box indicates a full adder.

Property 1. Let $(x, y, c_{in})$ be the inputs of a full adder, and $(s, c_{out})$ be the sum and carry outputs respectively. The full adder can be partitioned into two disjoint logic functions if, without loss of generality, we connect one bit $b$ to $x$ and $y$. Then $c_{out} = b$ and $s = c_{in}$.

For example, in Figure 3(b), we use two LUTs to route signal $a_0$ to two inputs of the first full adder in the chain, and set the carry-input to zero. Property 1 then allows this full adder to route $a_0$ to the carry-input of the next full adder in the chain; the sum output, which is zero, is not used.

Similarly, both GPCs must route the carry-output of the last full adder in the carry-chain to an ALM output; the sum output has a direct connection to the ALM output, but the carry output does not. Once again, we can exploit Property 1 by routing the carry output, for example, $z_2$ in Figure 3(b), to the carry-input next full adder in the chain. We configure the LUTs so that the other inputs to the full adder are both zero; this propagates $z_2$ to the ALM output through the sum output of the last adder in the chain. Moreover, this produces a carry-output of zero. This effectively *breaks* the carry-chain, so a new carry-chain (with carry-input zero) can start at the following full adder.

Figures 4(a) and 4(b) show a slice-based implementation of (0, 7; 3) GPC for the Virtex-5. Figure 4(a) depicts the design built using a network of full and half-adders, and Figure 4(b) shows the same design mapped onto one slice. The LUT-only implementation of this GPC requires two slices, because the GPC has 7 inputs, while a slice only has 6. The adder chain that is selected for the mapping to the carry-chain has been highlighted in both figures and the remaining adders are mapped to the driving LUTs. As mentioned in Section 3.2, part of the adder that is placed on the carry-chain is implemented by the LUT as shown in Figure 4(b). The other important feature of this implementation is that the carry output of the second LUT, $c_1$, is not dependent on

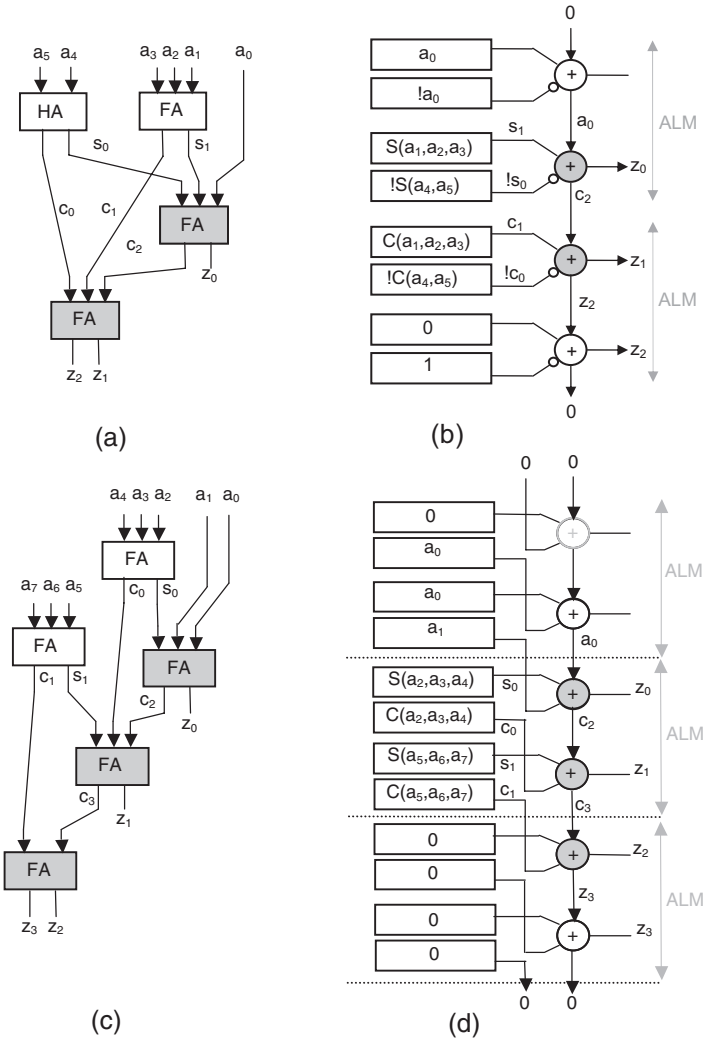


Fig. 3. A (0, 6; 3) GPC implemented at the circuit level (a) and synthesized on ALMs and carry-chains using Arithmetic Mode (b). A (3, 5; 4) GPC implemented at the circuit level (c) and synthesized on ALMs and carry-chains using Shared-Arithmetic Mode (d).

the output bit of the first LUT, $s_0$; this prevents the formation of a multi-LUT critical path involving carry-chains.

In Virtex-5, an input can access the carry-chain at any point, but the most significant output ($z_2$) goes through the last multiplexer of the chain. One additional quarter-slice is required to generate the GPC output. Since the multiplexer output drives an *xor* gate, the other input is set to constant "0" to propagate the last GPC output to the slice output.

We can map up to 8-input GPCs to the logic cells of Stratix III and Virtex-5 using the carry-chain. The main constraint is that no routing wires are used within each GPC.

*Definition* 1. A *covering GPC* is one whose functionality, given I/O constraints, cannot be implemented by another GPC.

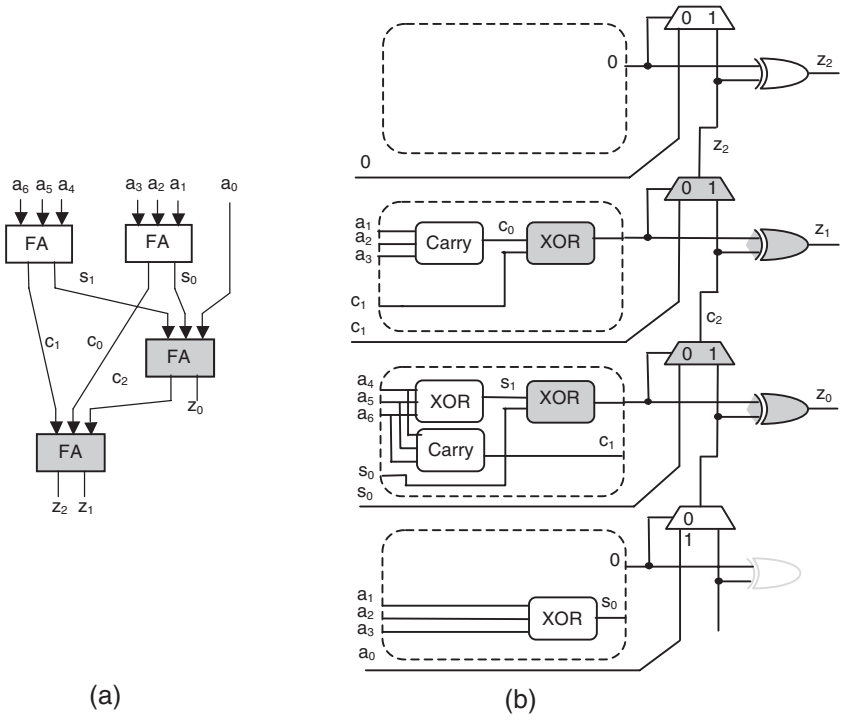


Fig. 4. A (0, 7; 3) GPC implemented at the circuit level (a) and synthesized on a Virtex-5 slice (b) using the carry-chain.

Table I. Covering GPC Libraries for the Stratix III (left) and Virtex-5 (right)

| | Altera Stratix III | | | | Xilinx Virtex-5 | | | |
|---|---|---|---|---|---|---|---|---|
| | LUT Only | | Arithmetic | | LUT Only | | Arithmetic | |
| GPCs | Delay (ns) | Area (ALMs) | Delay (ns) | Area (ALMs) | Delay (ns) | Area (LUTs) | Delay (ns) | Area (LUTs) |
| (0,6;3) | 0.38 | 3 | 0.97 | 2 | 0.35 | 3 | 1.04 | 4 |
| (1,5;3) | 0.38 | 3 | 0.97 | 2 | 0.35 | 3 | 0.79 | 3 |
| (2,3;3) | 0.38 | 3 | 0.97 | 2 | 0.35 | 3 | 0.79 | 3 |
| (0,7;3) | 1.36 | 4 | 0.98 | 2.5 | 1.48 | 6 | 1.04 | 4 |
| (1,6;4) | 1.36 | 5 | 1.01 | 3 | 0.84 | 7 | 1.04 | 4 |
| (3,5;4) | 1.36 | 5 | 1.01 | 3 | 0.65 | 7 | 1.04 | 4 |
| (4,4;4) | 1.36 | 5 | 1.01 | 3 | 0.91 | 6 | 1.04 | 4 |
| (5,3;4) | 1.36 | 5 | 1.01 | 3 | 0.65 | 5 | 1.04 | 4 |
| (6,2;4) | 1.36 | 5 | 1.01 | 3 | 0.91 | 7 | 1.04 | 4 |

For instance, a (4, 3; 4) GPC is *not* a covering GPC: either a (4, 4; 4) or (5, 3; 4) can implement its functionality by setting an appropriate input bit to zero.

The GPC library only contains covering GPCs. Table I summarizes the GPC libraries for the Stratix III and Virtex-5. Each GPC in the library can be implemented with only LUTs, or with LUTs in conjunction with carry-chains, as discussed before. When a noncovering GPC is needed during compressor tree synthesis, the smallest covering GPC that can implement its functionality is always chosen.

For the Stratix III, GPCs using Arithmetic Mode are uniformly smaller than those built using only LUTs. For GPCs with six or fewer inputs, the LUT-only implementation is faster. For GPCs with more than six inputs, two layers of LUTs are required for the LUT-only implementation, while Arithmetic Mode can realize the same GPC using a

single layer of LUTs in conjunction with a carry-chain; thus, the latter is faster and smaller.

The Virtex-5 GPC library has different characteristics. For GPCs with six or fewer inputs, the LUT-only implementations are uniformly superior to the use of carry-chains. For the (0, 7; 3) GPC, the carry-chain-based implementation is faster and smaller than the LUT-only implementation; for all remaining GPCs, the LUT-only implementations are faster, but larger, than the carry-chain-based implementations.

### 3.4. Efficiently Packing Adjacent GPCs Along Carry-Chains

Each Stratix III ALM, for example, contains ten ALMs, but LAB inputs can only enter the carry-chain between the first and sixth ALM, as discussed in Section 3.1. As shown in Figures 3(b) and (d), two GPCs can be abutted, because the carry-in and carry-out bits of each are not part of the GPC circuit. However, the placer in Altera's Quartus II software is unable to pack GPCs densely in a LAB, because it requires an explicit connection via a carry-chain from one GPC to the next. Quartus II only instantiates two GPCs per LAB: one starting at the first ALM, and one starting at the sixth. For example, if two (0, 6; 3) GPCs were synthesized, then just four of the ten ALMs in a LAB would be used.

Two GPCs that use the same configuration mode, for example, (Shared) Arithmetic Mode, can share an ALUT when abutted. Looking at Figure 3(b), the first ALUT in a GPC produces no output, and the last ALUT receives no inputs. Property 1 allows the last ALUT of one GPC to be shared with the first ALUT of the next, as shown in Figure 5(a). This allows a new GPC to start at any point along the carry-chain, not just at the first or sixth ALM, and facilitates resource sharing between adjacent GPCs. Fortunately, the Quartus-II's was able to discern that the two GPCs are logically disjoint.

Referring back to Table I, the GPCs with six or fewer inputs require 2 ALMs, but when $n$ such GPCs are abutted, one ALUT (half-ALM) is shared between each pair and therefore $3n + 1$ ALUTs are used. We abut groups of GPCs that use up to five contiguous ALMs (half-LAB) and 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.

The Virtex-5 offers a more limited opportunity to share LUT and carry-chain resources between abutted GPCs; this technique only works for the (0, 7; 3) and (2, 3; 3) GPCs in Table I. Figure 5(b) shows an example for two *(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.

## 4. COMPRESSOR TREE SYNTHESIS HEURISTIC

Given a GPC library, this section describes a heuristic to synthesize a compressor tree. The first step characterizes each GPC in the library in terms of its ability to reduce the number of bits at each stage. Second, a greedy heuristic generates the compressor tree.

### 4.1. GPC Library Characterization

The first step is to prioritize the GPCs in the library. We introduce three metrics for this purpose.

*Definition* 2. The *Compression Difference* (CD) of a GPC is the difference between the number of inputs and the number outputs.

*Definition* 3. The *Performance Degree* (PD) of a GPC is the ratio PD = CD/delay.

*Definition* 4. The *Area Degree* (AD) of a GPC is the ratio AD = CD/area.

*Definition* 5. The *Area-Performance Degree* (APD) is AD. PD.

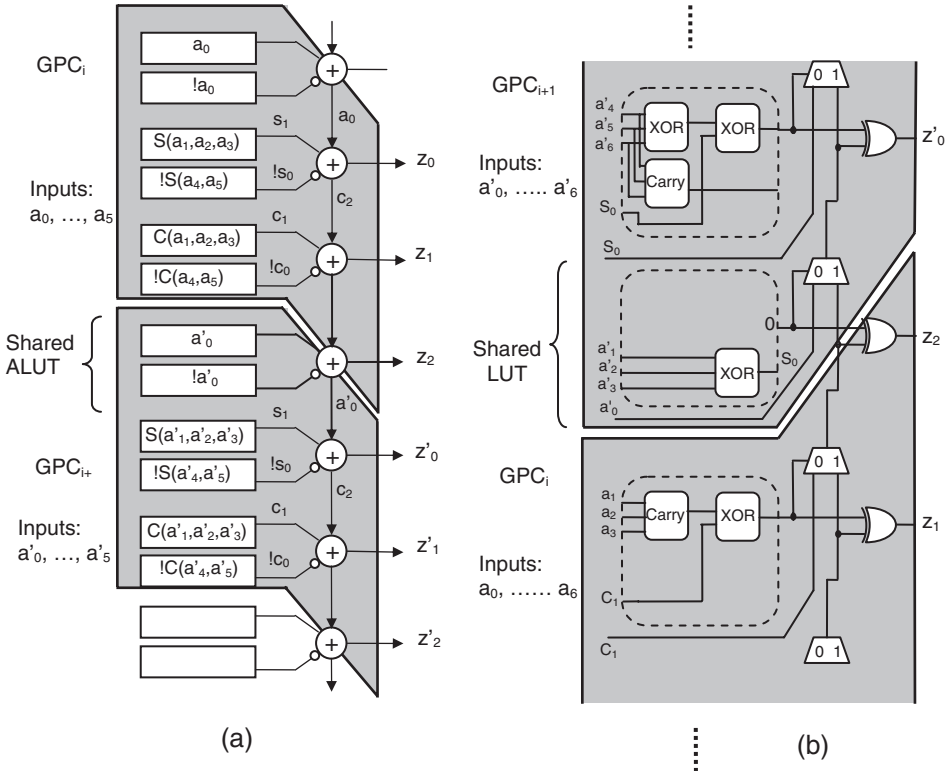


Fig. 5. Two abutted (0, 6; 3) GPCs on the Stratix III implemented using the same mode (Arithmetic or Shared Arithmetic) can share an ALUT, improving area utilization (a); two (0, 7; 3) GPCs on the Virtex-5 abutted using shared LUTs to reduce area. Only portions of both GPCs are shown to conserve space (b).

The compression difference represents each GPC's ability to reduce the bits at each level of the compressor tree. For example, the compression difference of a (2, 3; 3) GPC is $5 - 3 = 2$, while that of a (6, 2; 4) GPC is $8 - 4 = 4$; thus, the latter is more effective.

The three objective criteria outlined before 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. The PD criterion is used to optimize delay; AD is used to optimize area; and APD attempts to balance delay and area. Table II lists the CD value for each GPC, along with the PD, AD, and APD values resulted for each GPC listed in Table I. The GPCs having the highest priority for the different design objectives have been encircled with rectangles in Table II; these GPCs are called *base GPCs*.

### 4.2. Compressor Tree Synthesis Heuristic

The input to the compressor tree synthesis heuristic is a set of bits of different weights to sum. A *column* is a set of bits having the same weight, that is, *column[i]* is the number of input bits in the $i^{\text{th}}$ column. As an example, Appendix B describes the input representation of a FIR filter; for this particular filter, $column[0] = 1, column[1] = 3$, etc. Additionally, the user specifies one of PD, AD, or APD as the optimization strategy, and the GPCs in the library are sorted accordingly.

Coincidentally, all of the base GPCs in Table II are *m:n* counters. To exploit this fact, the heuristic first covers all of the bits in $column[i]$ with as many base GPCs as

Table II. The CD Value for Each GPC, and the PD, AD, and APD Values for the Stratix III and Virtex-5 GPC Libraries Listed in Table I

| | | Altera Stratix III | | | | | | Xilinx Virtex-5 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | LUT Only | | | Arithmetic | | | LUT Only | | | Arithmetic | | |
| GPC | CD | PD | AD | APD | PD | AD | APD | PD | AD | APD | PD | AD | APD |
| (0,6;3) | 3 | 7.9 | 0.9 | 7.1 | 3.1 | 1.8 | 5.6 | 8.5 | 1 | 8.5 | 2.9 | 0.6 | 1.7 |
| (1,5;3) | 3 | 7.9 | 0.9 | 7.1 | 3.1 | 1.8 | 5.6 | 8.5 | 1 | 8.5 | 3.8 | 1 | 3.7 |
| (2,3;3) | 2 | 5.3 | 0.6 | 3.2 | 2.1 | 1.2 | 2.5 | 5.5 | 0.6 | 3.3 | 2.5 | 0.6 | 1.5 |
| (0,7;3) | 4 | 2.9 | 0.8 | 2.3 | 4.1 | 2.9 | 8.2 | 1.2 | 0.6 | 0.7 | 3.8 | 1.3 | 4.8 |
| (1,6;4) | 3 | 2.2 | 0.6 | 1.3 | 3 | 1 | 3.0 | 3.6 | 0.4 | 1.4 | 2.9 | 0.7 | 2.0 |
| (3,5;4) | 4 | 2.9 | 0.8 | 2.3 | 4 | 1.6 | 6.3 | 6.2 | 0.5 | 3 | 3.8 | 1 | 3.8 |
| (4,4;4) | 4 | 2.9 | 0.8 | 2.3 | 4 | 1.6 | 6.3 | 4.4 | 0.6 | 2.6 | 3.8 | 1 | 3.8 |
| (5,3;4) | 4 | 2.9 | 0.8 | 2.3 | 4 | 1.6 | 6.3 | 6.2 | 0.8 | 4.9 | 3.8 | 1 | 3.8 |
| (6,2;4) | 4 | 2.9 | 0.8 | 2.3 | 4 | 1.6 | 6.3 | 4.4 | 0.5 | 2.1 | 3.8 | 1 | 3.8 |

The GPC with the highest priority in each case has been identified with a rectangle.

necessary; some bits may be left over. For example, if the base GPC is (0, 6; 3), and $column[i]$ contains *8* bits, then one base counter would be used to cover the first six bits, and two bits would be left over. The output bits of each GPC must propagate to the correct columns. For example, a (0, 6; 3) GPC that covers six bits in $column[i]$ will produce three output bits of rank $i$, $i+1$, and $i+2$; these bits must be added to *column[i]*, $column[i+1]$, and $column[i+2]$ accordingly.

The second step is to cover the remaining bits with counters. The heuristic traverses the columns from least to most significant, and selects an appropriate GPC for the library to cover each column. GPCs are considered in priority order, and the first feasible GPC is chosen. Columns containing three or fewer bits are skipped, because our target FPGAs support 3-input. To bind a GPC to $column[i]$, two conditions must be met. Firstly, a GPC with exactly $column[i]$ bits in its least significant position must be chosen. Secondly, the subsequent column(s) should have at least as many bits as GPC inputs in that position. For example, suppose $column[i] = 3$ and we are considering a (2, 3; 3) GPC; then $column[i+1] \geq 2$ to satisfy this criterion. After a GPC has been chosen, the bits that it covers are removed from their respective columns. The process then continues, starting with $column[i+1]$ and stopping at the most significant column.

The third step generates the output bits for each GPC; as discussed previously, an $M$-output GPC whose least significant inputs cover bits from $column[i]$ generates one output bit for $column[i], column[i+1], \ldots, column[i+M-1]$. Output bits are generated *after* the covering process stops; this prevents the formation of carry-chains at each level of the tree. For example, a GPC covering $column[i]$ will produce an output bit for $column[i+1]$; we want that bit to be covered at the *next* level of the tree, rather than the current level.

Next, we connect the GPCs from the current level of the tree to those at the previous level. It is therefore important to track which GPC produces each output bit. When LUT-based GPCs are used, each output bit of the GPC has the same delay; however, when carry-chains are used, the least significant output bit produced by each GPC has a slightly lower delay than the second least significant output bit, etc., for example, as illustrated by Figures 3(b) and 4(b) for the Stratix III and Virtex-5, respectively. Similarly, input-to-output delays of certain inputs may be higher than others; for example, in Figure 3(b), input bit $a_0$ clearly has the longest critical path. Thus, it is generally a good strategy to connect bits with higher arrival times to the GPC inputs with lower critical path delays at each level.

Both the Stratix III and Virtex-5 FPGAs support native 3-input carry-propagate addition through their carry-chains. If all columns contain three or fewer bits, then the compressor tree generation is complete, and all that remains is to connect the

Table III. Benchmark Summary

| Benchmark | Description |
|---|---|
| DCT | Multiplierless DCT |
| HPoly | Horner polynomial Eval. |
| H.264 ME | H.264 motion estimation |
| G.721 | G.721 encoder |
| FIR3, FIR6 | 3- and 6-tap FIR filters |
| SM9 × 9, SM18 × 18 | Parallel signed multipliers |
| vm.add2I, vm.add2Q | Video mixer |

compressor tree outputs to a carry-propagate adder of appropriate bitwidth. If at least one column contains four or more bits, then another compressor tree level is generated, using the same technique outlined earlier.

## 5. EXPERIMENTAL RESULTS

### 5.1. Experimental Methodology

First, we modeled each coveing GPC in Table I at a low-level granularity. For the Stratix III, we performed atom-level modeling using the *Verilog Quartus Module (VQM)* format, as provided by Altera's Quartus-II University Interface Program (QUIP). These models were used as components to construct larger compressor trees. The delay and area values reported here are taken from the Quartus-II project reports. For the Virtex-5, we took a similar approach, using a Verilog-like format similar to VQM. Xilinx's ISE 10.1 CAD tools were used for all experiments targeting the Virtex-5. The mapping heuristic was implemented in C++ using delay profiles for each GPC provided by the synthesizer. The input is a text file containing the number of bits per column. The output is a structural VHDL netlist of GPCs, forming the compressor tree, followed by a 3-input carry-propagate adder. The user specifies PD, AD, or APD at the command line.

### 5.2. Benchmarks

Table III summarizes the benchmarks used in our experiments, which are compressor trees taken from arithmetic circuits and DSP and video processing applications. DCT [Shams et al. 2000], H.264 ME [Chen et al. 2006], FIR3, FIR6 [Mirzaei et al. 2006], sm9x9, and sm18x18 naturally contain compressor trees. HPoly, G.721 [Lee et al. 1997], and Video Mixer [Synopsys 2001] are transformed to expose large compressor trees [Verma et al. 2008]. Most of these benchmarks are publicly available with a few exceptions. The FIR filters were built using randomly generated constants as described in Appendix-B; and Video Mixer is provided by Synopsys Corporation as an example to illustrate their Behavioral Optimization of Arithmetic (BOA) tool. Video mixer contains several distinct compressor trees and we use two of them. Each benchmark is synthesized as a purely combinational circuit, using four different approaches: *3-Add* uses a tree of three-input adders, *MaxPD*, *MaxAD*, and *MaxAPD* use the compressor tree synthesis heuristic with GPCs prioritized by PD, AD, and APD, respectively. The approaches are evaluated and compared in terms of critical path delay and area.

### 5.3. Results: Stratix III

Figures 6 and 7 respectively report the critical path delay and area for each benchmark using each of the four synthesis methods for Altera's Stratix III FPGA.

Figure 6 shows that that 3-Add has the maximum critical path delay for all benchmarks except *g721*. For *g721*, MaxAD has a slightly larger critical path delay than 3-Add, but the critical path delays of both MaxPD and MaxAPD, which include critical path delay as part of the GPC prioritization scheme, are significantly smaller. On

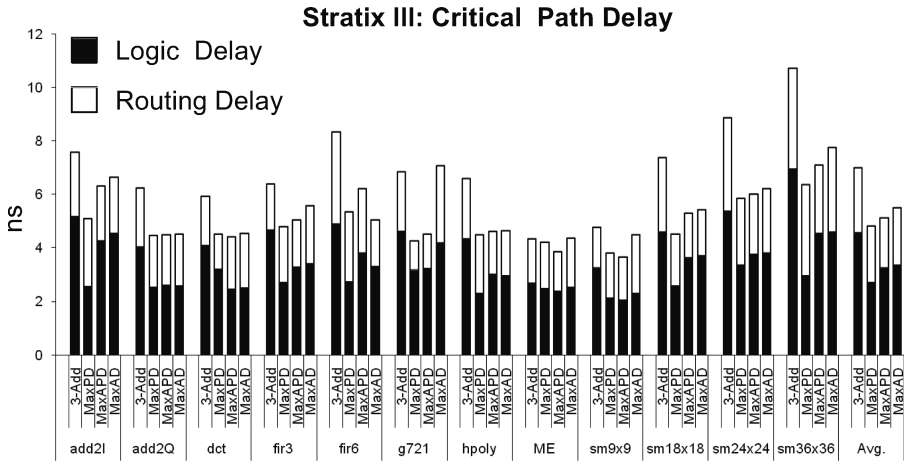


Fig. 6. The critical path delay of 3-ADD, MaxPD, MaxAPD, MaxAD decomposed into logic and routing delay after synthesis on the Stratix III.

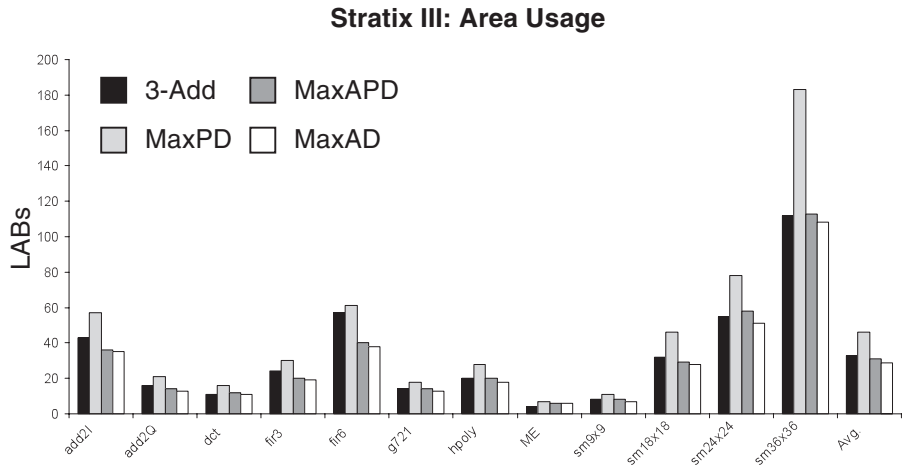


Fig. 7. Area usage (LABs) of the four synthesis methods on the Stratix III.

average, the critical path delay of 3-Add is 45% greater than the critical path delay of MaxPD.

Logic delays, that is, the delay through LUTs and carry-chains, rather than routing delay, are the primary reason that 3-Add has greater logic delay than the compressor trees; this is due, primarily, to the long ripple-carry-chains that are formed using the Stratix III ALM's Shared-Arithmetic Mode. Logic delay is more prominent for benchmarks having the greatest height, such as *add2I* and *sm*36x36. On the other hand, benchmarks such as *g721* and *ME* have shorter adder trees, but wide-bitwidth final carry-propagate adders; thus, they exhibit little disparity between adder and compressor trees in terms of delay.

MaxPD achieves the smallest critical path delay for most benchmarks, as it uses the LUT-only (0, 6; 3) base GPC having highest priority. The delay of this GPC is less than that of the base GPCs for MaxAPD and MaxAD; this explains MaxPD's advantage in terms of delay. On average, MaxAPD's critical path delay is 7.6% greater than MaxPD's.

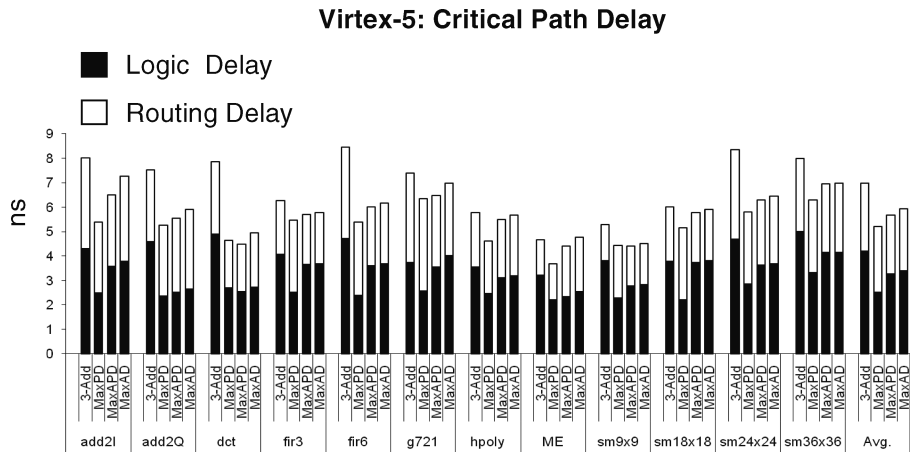


Fig. 8. The critical path delay of 3-ADD, MaxPD, MaxAPD, MaxAD decomposed into logic and routing delay after synthesis on the Virtex-5.

In terms of area, both MaxAPD and MaxAD require fewer LABs than 3-Add, most notably *fir3* and *fir6*. MaxPD uses more LABs than the other compressor tree synthesis methods, because its base (0, 6; 3) GPC has a large LUT-only implementation (3 ALMs), compared to the (0, 7; 3) base GPC of use by MaxAPD and MaxPD, which requires (2.5 ALMs) when implemented with carry-chains. 3-Add requires more LABs than MaxAPD and MaxAD because each LAB has limited input bandwidth, which inhibits the ability to use all ALMs in a LAB to implement a 3-input adder. Each LAB contains 10 ALMs, and 6 inputs per ALM are used in Shared-Arithmetic Mode, so 60 inputs are required to implement a 10-bit 3-input adder in a LAB. A closer analysis shows that only half of the ALMs in the LAB were used; in contrast, it is possible to fit six 6-input GPCs into a LAB, requiring only 36 inputs, which is below the LAB input bandwidth.

To summarize, MaxAPD offers the best trade-off between critical path delay and area usage. MaxPD should only be used when the compressor tree is constraining the critical path delay of the entire system and there are extra LABs to spare. MaxAD is clearly the best synthesis mode if area reduction is a priority.

### 5.4. Results: Virtex-5

Figures 8 and 9 respectively report the critical path delay and area for each benchmark using each of the four synthesis methods for Xilinx's Virtex-5 FPGA.

Like the Stratix III, 3-Add has the largest critical path delay for most benchmarks, although there are exceptions such as *dct* and *ME* where MaxAD and Add-3 are approximately equal. MaxPD achieves the smallest delay among all benchmarks other than *g721* and *sm*9 $\times$ 9. The disparity of the critical path delays between MaxPD, MaxAPD, and MaxAD is smaller for the Virtex-5 than the Stratix III; this is due to architectural features, such as different carry-chains and the fact that the Virtex-5 does not have LAB-like clusters of LUTs with fast local routes. On average, MaxPD is 33%, 8.8%, and 14% faster than 3-Add, MaxAPD, and MaxAD, respectively.

Like the Stratix III, 3-Add has the largest logic delay, due to ripple-carry propagation. The base GPC for MaxPD is a (0, 6; 3) GPC implemented using LUTs alone, while MaxAPD and MaxAD use a (0, 7; 3) GPC implemented using LUTs and carry-chains as their base GPCs; thus, MaxAPD and MaxAD have larger logic delays than MaxPD.

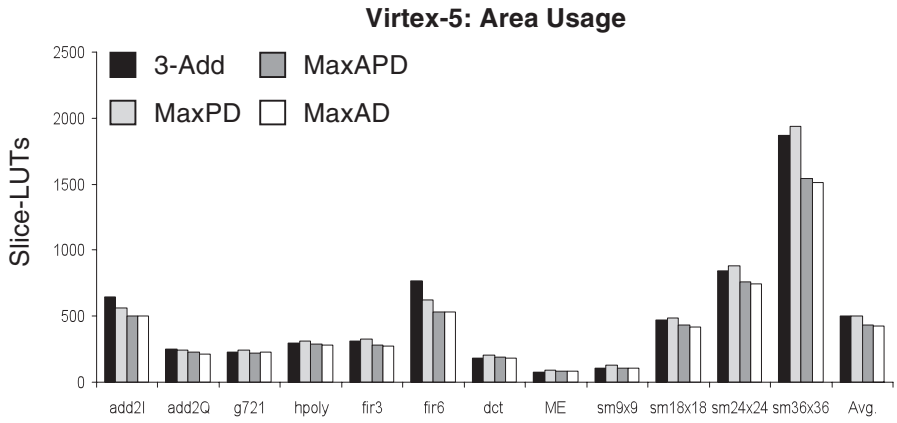


Fig. 9. Area usage (LABs) of the four synthesis methods on the Virtex-5.

The area results are somewhat different than the Stratix III; particularly, MaxPD does not suffer from particularly poor area utilization for the Virtex-5. On average, 3-Add and MaxPD are comparable in terms of area (and thus, MaxPD is preferable due to its reduced critical path delay); while MaxAPD and MaxAD achieve less pronounced, but noticeable, average improvements in area, while remaining comparable to one another.

To summarize, critical path delay, and not area, distinguishes the four synthesis heuristics for the Virtex-5. 3-Add is not competitive in terms of critical path delay for the Virtex-5, while offering no area advantage, unlike the Stratix III. On average MaxPD achieves slightly better critical path delays than MaxAPD and MaxAD, but uses slightly more area; MaxAPD and MaxAD are comparable in terms of both critical path delay and area. If the compressor tree lies on the critical path of a design, then MaxPD should be used; otherwise, MaxAPD or MaxAD would be more appropriate choices.

### 5.5. Integer Linear Programming (ILP)

To assess the quality of the heuristic, we compared it with an Integer Linear Program (ILP) that computes a near-optimal compressor tree implementation. Ideally, one could formulate the ILP to exploit the GPC delay and area information reported in Table I, combined with all packing possibilities discussed in Section 3.4 to obtain an overall optimal solution for any of the three design objectives; however, we found that the search space is too large and that the ILP does not converge in a reasonable amount of time. Instead, we modified a previous ILP [Parandeh-Afshar et al. 2008b] to use the base GPCs specified in Table II (for the three respective objectives) to reduce the size of the tree until each column contains no more than six (MaxPD) or seven (MaxAPD, MaxAD) bits.

This limits the size of the search space for the initial stages of the compressor tree, which improves runtime at the expense of optimality. The remainder of the ILP tries to minimize the number of GPCs and the number of resulting bits produced by those GPCs, in accordance with the prior formulation [Parandeh-Afshar et al. 2008b]. On average, the ILP improved the delay by less than 0.5%, and LUT usage by approximately 2%. These results indicate that the heuristic produces solutions of good quality, without suffering the high runtime overhead of the ILP.

## 6. RELATED WORK

### 6.1. Compressor Tree Synthesis for FPGAs

Conventional wisdom has held that adder trees are superior to compressor trees on FPGAs. For example, Altera's manual for the Stratix II notes that the Shared-Arithmetic Mode of the ALM was introduced to facilitate implementation of adder trees using 3-input, rather than 2-input, adders [Altera Corp. 2011], which reduces the height of an $N$-input adder tree from $\lceil log_2 N \rceil$ to $\lceil log_3 N \rceil$. Our experiments have shown that compressor trees can be more effective than adder trees in practice, and that Shared-Arithmetic Mode can actually be quite useful for constructing compressor trees.

Poldre and Tammemae [1999] developed a method to synthesize a 4:2 compressor on the Xilinx Virtex slice architecture; a layer of 4:2 compressors can reduce four integers to two without carry propagation [Weinberger 1981]. Similarly, 4:2 compressors have also been synthesized on the slice architecture used in the Xilinx Virtex-2 and -4 and Spartan-2 and -3 [Ortiz et al. 2009; Kamp et al. 2009], Altera Cyclone III [Kamp et al. 2009], and Altera's Adaptive Logic Module (ALM) [Paidimarri et al. 2009]. A 4:2 compressor only requires 4 input bits per LUT, which is a good implementation choice for older and lower-end FPAs whose logic cells are based on 4-LUTs with carry-chains; however, as modern FPGAs now contain fracturable 6-LUTs [Hutton et al. 2004] better I/O utilization can be achieved using the larger GPCs advocated in this article.

Three prior papers have proposed to synthesize compressor trees on FPGAs using LUT-based GPCs only [Parandeh Afshar et al. 2008a, 2008b; Matsunaga et al. 2010]. The experimental results in this article indicate that this approach improves critical path delay, but increases the area significantly, and, as a consequence, may not yield an ideal design choice. This article is an extension of a more recent work [Parandeh-Afshar et al. 2009] which was the first to consider the design of GPCs that exploit the Shared-Arithmetic Mode of Altera's ALMs. The extensions in this article are twofold: (1) the approach has been extended to consider the carry-chains used in Xilinx's Virtex-5 FPGAs; and (2) presents a simplified mapping heuristic that produces comparable results.

### 6.2. Compressor Tree Synthesis for ASICs

Compressor trees were introduced as an efficient method for partial product reduction for parallel multipliers using networks of full and half-adders [Wallace 1964; Dadda 1965]. This was soon followed by the notion of parallel counters [Swartzlander, Jr. 1973] and GPCs [Stenzel et al. 1977], which are constructed using full and half-adders as building blocks. An optimal compressor tree synthesis algorithm, which repeatedly chooses the three bits from each column having the smallest arrival times and connects them to the input of a full adder, was introduced more recently [Stelling et al. 1998; Um and Kim 2002]. The design of the final adder is then tailored to the delay profile of the compressor tree outputs [Oklobdzija and Villeger 1995; Stelling and Oklobdzija 1996]. These approaches do not work well for FPGAs for two reasons: (1) routing delays are difficult to predict during synthesis; and (2) the presence of carry-chains influences the structure of both GPCs and the final adder.

## 7. CONCLUSION

This article introduces a new approach for the synthesis of compressor trees on commercial high-performance FPGAs. Unlike prior work on compressor tree synthesis, this approach is able to exploit the carry-chains that are present in FPGA logic cells, which were originally designed for carry-propagate, rather than carry-save, addition.

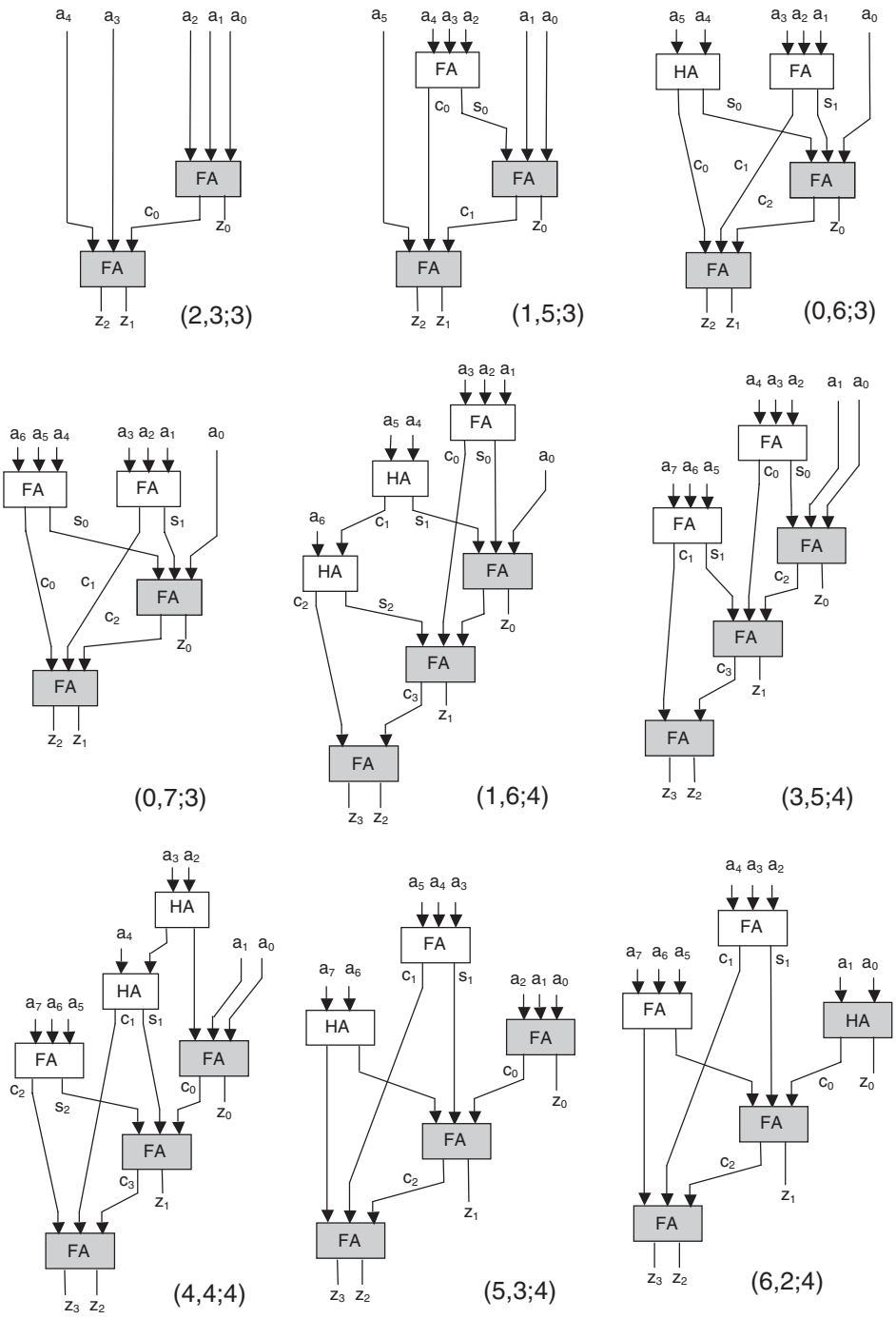


Fig. 10. The covering GPCs listed in Tables I and II as networks of full and half-adders. The shaded full and half-adders are synthesized on the carry-chains.

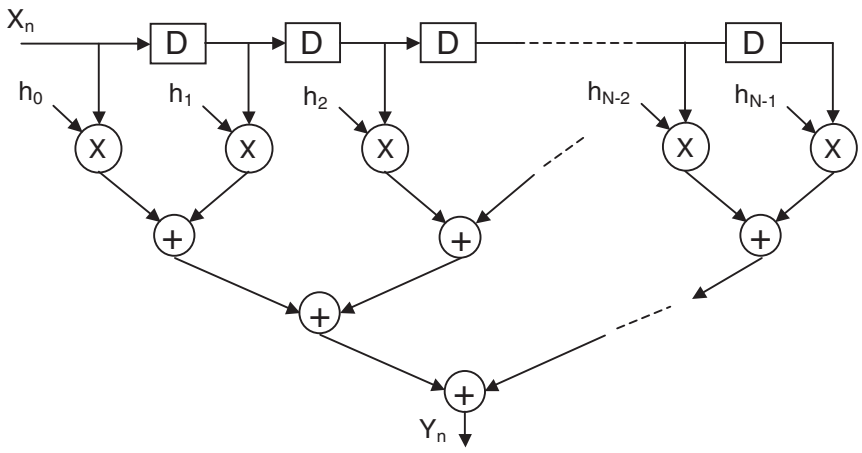


Fig. 11. The structure of an N-length FIR filter.

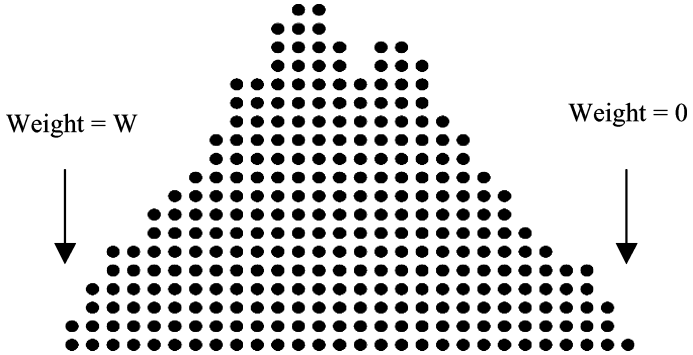


Fig. 12. The dot-notation representation of the FIR3 benchmark.

The approach involves two stages: an architecture-specific GPC library instantiation phase, which is currently performed by hand (but only once, per target architecture); and an architecture-agnostic compressor tree synthesis phase, which constructs a compressor tree using components from the library. Our results targeting both FPGAs from Altera and Xilinx indicate that compressor trees can improve both critical path delay and area compared to the existing state-of-the-art: synthesis of multioperand addition using trees of 3-input adders. Altogether, the compressor tree synthesis algorithm presented here can help users and CAD tool developers improve the arithmetic capabilities of modern high-performance FPGAs.

## APPENDIXES

### APPENDIX A: LOGICAL STRUCTURES OF COVERING GPCs

Figure 10 shows the all covering GPCs of Table I/II as networks of full and half-adders. The highlighted adders are mapped to the FPGA carry-chains and the others are mapped to the driving LUTs based on the design approach that was described in Section 3.3.

## APPENDIX B: FIR FILTER IMPLEMENTATION

A *Finite Impulse Response* (FIR) filter implements a digital frequency response. A FIR filter is typically implemented using a series of delays, multipliers, and adders. Figure 11 depicts a length-N FIR filter. The delays allow the filter to operate on a sequence of samples, and the $h_k$ coefficients are used for multiplication. The output at time $n$ is the sum of the delayed samples multiplied by the coefficients. Thus, ignoring the delays, the entire circuit can be implemented as a large compressor tree. The usage of constant multipliers causes the shape of the compressor tree to be irregular, and its precise shape depends on the constant values chosen. Figure 12 depicts the compressor tree of a length-3 FIR filter using dot notation, where each dot depicts in a bit in a specific column that will be summed.

The number of bits per column and the number of columns are the inputs to the compressor tree synthesis heuristic described in Section 4.2.

## REFERENCES


ALTERA CORPORATION. Stratix II, III, and IV device handbooks. http://www.altera.com/.

DADDA, L. 1965. Some schemes for parallel multipliers. *Alta Frequenza 34*, 349–356.

HUTTON, M., SCHLEICHER, J., LEWIS, D. M. PEDERSON, YUAN, S., KAPTANOGLU, S., BAECKLER, G., RATCHEV, B., PADALIA, K., BOURGEAULT, M., LEE, A., KIM, H., AND SAINI, R. 2004. Improving FPGA performance and area using an adaptive logic module. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications*. 135–144.

KAMP, W., BAINBRIDGE-SMITH, A., AND HAYES, M. 2009. Efficient implementation of fast redundant number addrs for long word-lengths in FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology*. 239–246.

MATSUNAGA, T., KIMURA, S., AND MATSUNAGA, Y. 2010. Multi-Operand adder synthesis on FPGAs using generalized parallel counters. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference*. 337–342.

OKLOBDZIJA, V. G. AND VILLEGER, D. 1995. Improving multiplier design by using improved column compression tree and optimized final adder in CMOS technology. *IEEE Trans. VLSI Syst. 3*. 292–301.

ORTIZ, M., QUILES, F., JORMIGO, J., JAIME, F. J., VILLALBA, J., AND ZAPATA, E. L. 2009. Efficient implementation of carry-save adders in FPGAs. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures, and Processors*. 207–210.

PAIDIMARRI, A., CEVRERO, A., BRISK, P., AND IENNE, P. 2009. FPGA implementation of a single-precision floating-point multiply accumulator with single-cycle accumulation. In *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 267–270.

PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. 2008a. Efficient synthesis of compressor trees on FPGAs. In *Proceedings of the Asia-South Pacific Design Automation Conference*. 138–143.

PARANDEH-AFHSAR, H., BRISK, P., AND IENNE, P. 2008b. Improving synthesis of compressor trees on FPGAs via integer linear programming. In *Proceedings of the International Conference on Design Automation and Test in Europe*. 1256–1262.

PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. 2009. Exploiting fast carry-chains of FPGAs for designing compressor trees. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*. 242–249.

POLDRE, J. AND TAMMEMAE, K. 1999. Reconfigurable multiplier for Virtex FPGA family. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. 359–364.

STELLING, P. F., MARTEL, C. U., OKLOBDZIJA, V. J., AND RAVI, R. 1998. Optimal circuits for parallel multipliers. *IEEE Trans. Comput. 47*, 273–285.

STELLING, P. F. AND OKLOBDZIJA, V. J. 1996. Design strategies for optimal hybrid final adders in a parallel multiplier. *J. VLSI Signal Process. 14*, 321–331.

STENZEL, W. J., KUBITZ, W. J., AND GARCIA, G. H. 1977. A compact high-speed parallel multiplication scheme. *IEEE Trans. Comput. C-26*, 948–957.

SWARTZLANDER JR., E. E. 1973. Parallel counters. *IEEE Trans. Comput. C-22*, 1021–1024.

UM, J. AND KIM, T. 2002. Layout-aware synthesis of arithmetic circuits. In *Proceedings of the 39th Design Automation Conference*. 207–212.

VERMA, A. K., BRISK, P., AND IENNE, P. 2008. Data-flow transformations to maximise the use of carry-save representation in arithmetic circuits. *IEEE Trans. Comput.-Aided Des.* 27, 1761–1774.

WALLACE, C. S. 1964. A suggestion for a fast multiplier. *IEEE Trans. Elec. Comput. 13*, 14–17.

WEINBERGER, A. 1981. A 4:2 carry save adder module. *IBM Tech. Disclos. Bull. 23*.

XILINX CORPORATION. Virtex 4, 5, and 6 device handbooks. http://www.xilinx.com.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30$^{th}$ International Symposium on Microarchitecture*. 330–335.

MIRZAEI, S., HOSANGADI, A., AND KASTNER, R. 2006. FPGA implementation of high speed FIR filters using add and shift method. In *Proceedings of the International Conference on Computer Design*. 308–313.

CHEN, C.-Y., CHIEN, S.-Y., HUANG, Y.-W., CHEN, T.-C., WANG, T.-C., AND CHEN, L.-G. 2006. Analysis and architecture design of variable block-size motion estimation for H.264/AVC. *IEEE Trans. Circ. Syst. 53*, 578–593.

SHAMS, A., PAN, W., CHANDANANDAN, A., AND BAYOUMI, M. 2000. A high-performance 1D-DCT architecture. In *Proceedings of IEEE International Symposium on Circuits and Systems*. 521–524.

SYNOPSYS. 2001. Creating high-speed data-path components—Application note. Tech. rep. Mountain View, CA, version 2001.08.

Received June 2010; revised December 2010; accepted March 2011