Improving Synthesis of Compressor Trees on FPGAs via Integer Linear Programming

Hadi Parandeh-Afshar[1,2]
[1]School of Electrical and Computer Engineering
University of Tehran
Tehran, Iran
e-mail : hparande@ut.ac.ir

Philip Brisk[2] and Paolo Ienne[2]
[2]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
e-mail : {philip.brisk, paolo.ienne}@epfl.ch

Abstract

Multi-input addition is an important operation for many DSP and video processing applications. On FPGAs, multi-input addition has traditionally been implemented using trees of carry-propagate adders. This approach has been used because the traditional look-up table (LUT) structure of FPGAs is not amenable to compressor trees, which are used to implement multi-input addition and parallel multiplication in ASIC technology. In prior work, we developed a greedy heuristic method to map compressor trees onto the general logic of an FPGA using a component called generalized parallel counter (GPC). Although this technique reduced the combinational delay of our circuits, when synthesized onto Altera Stratix-II FPGAs, by 27% on average; however, the area was increased by an average 11%. To further reduce the delay and limit the increase in area, we have developed a new solution to the mapping problem based on integer linear programming. This new approach reduced the delay of the compressor tree by 32% on average and reduced the area by 3% compared to an adder tree.

1. Introduction

The performance gap between FPGAs and ASICs is exacerbated for arithmetic circuits. One of the most important arithmetic operations in many DSP and video processing applications is multi-operand addition, i.e., the addition of $k > 2$ binary integers. Multi-input addition occurs in the context of FIR filters [1], correlation of 3G wireless base-station channel cards [2], motion estimation in video coding [3], and partial product summation in parallel multiplication [4, 5, 6, 7, 8-10, 11]. Verma and Ienne [12] developed a set of circuit transformations that can expose large compressor trees from disparate addition and multiplication operations. FPGAs can benefit significantly from techniques that can accelerate the performance of such operations.

Historically, multi-operand addition on FPGAs has been implemented as trees of *carry-propagate adders (CPAs)*; specialized circuitry has been added to commercial FPGAs by Xilinx and Altera to facilitate these efficient adder trees. In ASIC design, on the other hand, multi-operand addition is typically implemented as a compressor tree (e.g., a Wallace or Dadda Tree [4, 11]) built from *carry-save adders (CSAs)*. Due to the structure of FPGA architectures, adder trees have historically yielded better results than compressor trees in terms of both area and delay.

Previously [13], we developed a novel method to synthesize compressor trees onto FPGAs. This method uses *generalized parallel counters (GPCs)*, introduced in Section 2.2, as building blocks. Compared to CSAs and single-column parallel counters (Section 2.1), GPCs offer greater flexibility and increased logic and input utilization when synthesized on FPGAs. Here, an important question is how to best select GPC configurations and wire a collection of GPCs together to build a compressor tree. In [13], a greedy heuristic was used to construct a compressor tree. On average, the heuristic reduced the combinational delay of multi-operand addition on *Altera Stratix-II FPGAs* by 27% compared to adder trees; however, the *ALM (Adaptive Logic Module)* count increased by 11%. Although we were pleased with the reduction in delay, we were concerned about the area increase.

To address this concern, this paper reformulates the problem as an *Integer Linear Program (ILP)*, which is then solved using a commercial tool. Although solving ILPs is NP-Complete, our formulation includes additional constraints that can reduce the runtime of the solver, albeit at the expense of global optimality of the solution. The ILP finds solutions whose area is reduced by 3% on average compared to an adder tree. By using special constraints, the ILP reduces the arrival time of the least significant input bits to the final adder. Moreover, reducing the area yields a tighter placement of LUTs; this reduced the average interconnection length, which in turn, reduced wire delay. Altogether, the average combinational delay of the compressor tree synthesized by the ILP was reduced by 32% compared to the adder tree. Thus, the ILP demonstrates that it is possible to synthesize compressor trees on FPGAs that significantly reduce the delay and marginally reduce the area compared to adder trees.

2. Preliminaries

This section introduces the arithmetic components that are used to construct compressor trees, and describes the difference between compressor trees and adder trees.

Let $B = (b_{k-1}b_{k-2} \ldots, b_1b_0)$ be a k-bit binary number, where b_{k-1} is the *most significant bit (MSB)* and b_0 is the *least significant bit (LSB)*. The value of B is computed as follows:

$$B = \sum_{i=0}^{k-1} b_i 2^i \ . \qquad (1)$$

978-3-9810801-3-1/DATE08 © 2008 EDAA

The *rank* of a bit *b* determines its location (from rank *0*, the LSB, to rank *k-1*, the MSB) in an unsigned binary integer. If bit *b* has *rank i*, then *b* represents the quantity $b_i 2^i$. For example, the rank of each bit b_i in the above example is *i*.

If there are *n b*-bit unsigned binary integers to sum, a *column* refers to a set of bits having the same rank. For example, the ith column contains all of the bits of rank *i*.

2.1. Single-Column Parallel Counters

A *Single Column Parallel Counter* is a combinational circuit that accepts *m* input bits, counts the number of bits that are set to 1, and outputs that number as an *n*-bit unsigned binary number. Since the output must express a value in the range *[0, R]*, the number of bits required is

$$n = \lceil log_2(R+1) \rceil. \quad (2)$$

Henceforth, we will use the term *m:n counter* to refer to a single-column parallel counter with *m* inputs and *n* outputs.

In literature on arithmetic circuits, *2:2* and *3:2* counters are respectively called *half-adders (HAs)* and *full-adders (FAs)*. FAs and HAs are building blocks for binary integer adders (e.g. ripple-carry, carry-select, etc.). In the context of compressor trees, a *full-adder* is also called a *carry-save adder (CSA)*.

2.2. Generalized Parallel Counters

In an *m:n* counter described in the previous section, all *m* input bits have rank 0. A *generalize parallel counter (GPC)* is similar in principle to a counter, but it computes the sum of bits having different ranks. A GPC can be defined by the number of input bits of each rank and the number of outputs. We use the following notation to define a GPC: $(m_{k-1}, m_{k-2}, \ldots, m_1, m_0; n)$, $m_{k-1} > 0$, where m_i is the number of input bits of rank *i* and *n* is the number of output bits. The output of a GPC is an unsigned integer in the range *[0, R]*, where

$$R = \sum_{i=0}^{k-1} m_i 2^i \text{, and} \quad (3)$$

$$n = \lceil log_2(R+1) \rceil. \quad (4)$$

As an example, a *(5, 5; 4)* GPC is a GPC that takes 5-input bits of rank *0* and 5 input bits of rank *1*. The maximum output range, *R*, occurs when all input bits are 1, i.e., $R = 5 \cdot 2^1 + 5 \cdot 2^0 = 15$. Thus $n = 4$ output bits are required.

Fig. 1 illustrates the construction of compressor trees using both CSAs and GPCs. With CSAs, as shown in Fig. 1(a), the input bits are reduced to two rows; with a (3, 4; 4) GPC, in Fig. 1(b), alternatively, the input bits are reduced to a single row.

In ASIC design, a GPC can be constructed from single-column parallel counters; this limits its usefulness in that specific context. For example, the GPC in Fig. 1(b) can be constructed from FAs and HAs, as shown in Fig. 2. Observe that the two FAs shown on the top level of Fig. 2 are the same FAs shown in Fig. 1(a).

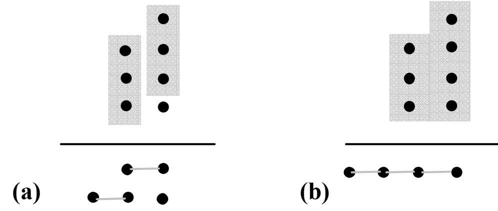

Figure 1. Compressor tree mapping using carry-save adders (a) and GPCs (b).

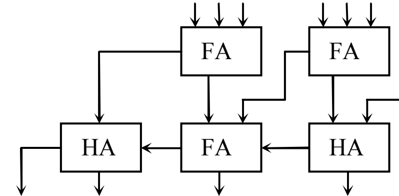

Figure 2. The GPC in Fig. 1(b) built from FAs and HAs.

On the other hand, a GPC can easily be implemented with LUTs, which makes it ideal building block for FPGA-based designs. For state-of-the-art high-performance FPGAs, e.g., the *Altera Stratix-II/III* and *Xilinx Virtex-4/5*, LUTs have 6 inputs; on earlier FPGAs, LUTs typically had 4. We limit the number of inputs of GPCs to 6 input bits so that they are implemented using only one logical layer of LUTs. A 6-input GPC with *n* output bits is synthesized on exactly *n* LUTs, with no dependencies.

2.3. Compressor Trees

Suppose that we want to compute the sum of *k n-bit* unsigned binary integers: $A_{k-1}, A_{k-2}, \ldots, A_1, A_0$. One approach would be to employ a tree of binary *n*-bit *carry-propagate adders (CPAs)*. A more effective approach, however, is to use a circuit called a *compressor tree*. A compressor tree computes two values, *Sum (S)* and *Carry (C)*, such that

$$S + C = \sum_{i=0}^{k-1} A_i \text{ .} \quad (5)$$

To compute the final sum, $S + C$ is computed using a CPA.

The Stratix-II/III and Virtex-5 FPGAs offer support for ternary addition. To exploit this fact, a faster compressor tree can be built that produces three outputs rather than two. If O_1, O_2, and O_3 are these outputs, then Eq. (5) is rewritten as

$$O_1 + O_2 + O_3 = \sum_{i=0}^{k-1} A_i \text{ .} \quad (6)$$

3. Related Work

We trace related work in two areas: techniques to construct compressor trees (Section 3.1) and efficient implementation of multi-operand addition on FPGAs (Section 3.2).

3.1. Constructing Compressor Trees

Techniques to build compressor trees from CSAs were proposed in the 1960s by Wallace [11] and Dadda [4]. In the context of parallel multipliers, where input bits to the compressor tree have different arrival times, Stelling et al. [8] described an optimal algorithm for compressor tree construction called *3-greedy (3GD)*. A layout-aware synthesis method that tries to minimize wirelength between CSAs was proposed by Um and Kim [9]. All of these methods used *2:2* and *3:2* counters building blocks.

Verma and Ienne [10] used an ILP to construct a compressor tree using a set of counters ranging from 2-8 input bits. This method is similar to ours in its use of larger counters. *4:2*, *5:2*, and *5:3* compressors (which are not counters) have also been used successfully in the design of multipliers [5]. They are similar to GPCs in that their input bits have different ranks; however, they also have carry-in and carry-out bits. Their connection structure, at first, looks like ripple-carry addition; however, each carry-out bit only ripples to one or two adjacent compressors before the rippling ends. The GPCs, in contrast, do not employ any rippling.

Mora et al. [6] constructed a compressor tree using ROMs called *LUT-Counters* to perform functionality similar to GPCs. A (p_t, q) LUT-Counter takes p columns, each containing t bits, and compresses them, providing q output bits. In our nomenclature, this would be a $(p_{t-1}, p_{t-2}, \ldots, p_1, p_0; q)$ GPC, where each $p_i = p$. Our work is more flexible in that we do not restrict each input column of the GPC to have the same number of bits. Secondly, their work targeted ASIC design, while we focus on FPGAs.

3.2. Multi-Operand Addition on FPGAs

Early FPGAs initially implemented all logic functions on 4-input LUTs, which did not yield particularly good arithmetic circuits, especially due to high routing delays. To speed up addition and shifting operations, fast carry-chains that connect adjacent LUTs without going through the programmable routing network were added to the basic LUT structure. Altera included FAs connected in ripple-carry fashion [14], and Xilinx added *xor* gates and programmable multiplexers to the fast carry-chains [15]; this allows the chains to be used for fast integer addition.

Poldre and Tammemae [7] constructed a compressor tree for parallel multipliers from *4:2 compressors* and synthesized them on *Xilinx Virtex* FPGAs, exploiting the fast carry-chains described above. They reported delays that were 1.5x faster and used 1.28x less area than standard adder trees.

FPGA architectures, however, have been enhanced since this work, and it is not clear whether these older techniques still suffice. In particular, the LUT size has been increased from 4 to 6 inputs, and both Altera (Stratix-II/III) and Xilinx (Virtex-5) have developed methods to use LUTs in conjunction with the fast carry-chains to perform ternary addition; the advantage of ternary addition was outlined in [16].

Previously [13], we successfully mapped compressor trees onto FPGAs using GPCs. The mapping technique used a greedy heuristic. This paper, in contrast, uses an ILP formulation that reduces both the average area and delay of the resulting circuit.

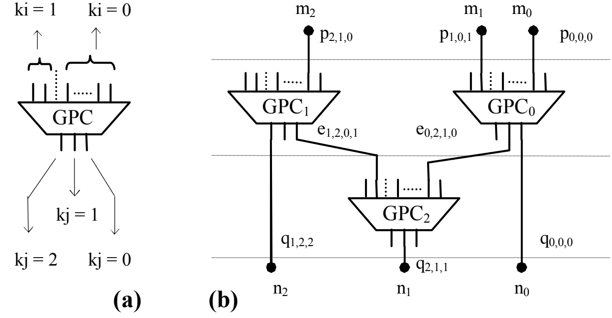

Figure 3. Example illustrating some of the variables used in the ILP.

4. ILP Formulation

This section outlines an *integer linear program (ILP)* that maps compressor trees onto a set of GPCs with at most 6 input bits of rank 0 or 1 and 3 output bits. A limit of 6 input bits per GPC was chosen since LUTs on modern FPGAs have 6 inputs. Constraints on the ranks and number of output bits were chosen to increase the compression realized with a fixed number of inputs and also limit the search space to increase the convergence rate of the ILP.
The input to the ILP is a set of unsigned binary integers whose sum is to be computed. Rather than represent each bit explicitly, we simply provide the number of columns, and the number of bits per column, to be summed. The output of the ILP is the configuration of each GPC (since there are multiple GPCs having 6-inputs and 3-outputs) and the connections between inputs, outputs, and GPCs.

4.1. Variables

Here, we introduce the variables that will be used in the ILP formulation:

M – The number of input bits.

N – The maximum number of GPCs.

$Inuse_i \in \{0, 1\}$ – True if GPC_i is in use; False otherwise.

$Level_i \in \{0, 1, \ldots, L\}$ – The logic level of GPC_i.

$Inum_0_i$ (Integer) – The number of connections to inputs of rank 0 of GPC_i.

$Inum_1_i$ (Integer) – The number of connections to inputs of rank 1 of GPC_i.

$p_{m,i,ki} \in \{0, 1\}$ – True if there is a connection between the mth input bit and an input of rank ki of GPC_i.

$q_{i,ki,m} \in \{0, 1\}$ – True if there is a connection between the ki-th output of GPC_i and an output bit of rank m.

$e_{i,j,ki,kj} \in \{0, 1\}$ – True if there is a connection from the ki-th output of GPC_i and an input of rank kj of GPC_j.

$D_{i,j} \in \{0, 1\}$ – True if there is a direct connection from the ith input bit and an output bit of rank j. These are input bits that are not connected to any GPC, but are instead connected directly to one of the inputs of the final ternary adder.

Fig. 3 provides an example showing some of the variables in the ILP.

4.2. Objective Function

As stated above, $Level_i$ is the logic level of GPC_i. $Level_i = 0$ if each input of GPC_i is either disconnected (i.e., set to 0) or is connected to one of the M input bits. Otherwise, let *preds(i)* be the set of GPCs whose outputs are connected to inputs of GPC_i. Then:

$$Level_i = \max_{j \in preds(i)} \left(Level_j \right) + 1 . \quad (7)$$

The objective function attempts to minimize the number of logic levels in the compressor tree:

$$Minimize : \max_{1 \le i \le N} \left(Level_i \right). \quad (8)$$

4.3. Input and Output Bit Constraints

Constraint (9) ensures that each input bit m is either connected to the input of exactly one GPC with the appropriate rank, or connected directly to an output column of the appropriate rank:

$$\sum_i \sum_{ki} p_{m,i,ki} + \sum_{n=rank(m)} D_{m,n} = 1 . \quad (9)$$

Since the Altera Stratix-II/III and Xilinx Virtex-5 FPGAs all support ternary addition, the compressor tree should produce an output having no more than *3* bits in each column. Constraint (10) enforces this restriction:

$$\sum_i \sum_{ki} q_{i,ki,n} + \sum_n D_{m,n} \le 3 . \quad (10)$$

4.4. Input Constraints for each GPC

No GPC can have more than 6 inputs. The number of inputs of rank 0 and rank 1 for the GPC_j, are $Inum_0_j$ and $Inum_1_j$ which are computed as follows:

$$Inum_0_j = \sum_{i \ne j} e_{i,j,ki,0} + \sum_{m=0}^{M} p_{m,j,0} \text{ , and} \quad (11)$$

$$Inum_1_j = \sum_{i \ne j} e_{i,j,ki,1} + \sum_{m=0}^{M} p_{m,j,1} . \quad (12)$$

Three input constraints are present. Constraint (13) ensures that at most 6 inputs are connected to each GPC that is in use. This ensures that each GPC can be implemented with one 6-input LUT per output bit, and that each GPC comprises one layer of logic.

$$Inum_0_j + Inum_1_j \le 6 \times Inuse_j . \quad (13)$$

Constraint (14) ensures that the sum of the input bits (weighted by rank) does not exceed 7, the maximum unsigned integer that can be represented with 3 output bits:

$$Inum_0_j + 2 Inum_1_j \le 7 \times Inuse_j . \quad (14)$$

Constraint (15) ensures that at least 5 input bits are connected to each GPC that is in use. The purpose of this constraint is to increase logic density, and therefore reduce the overall number of GPCs. It is important to note that the objective function does not try to optimize the utilization of GPCs.

$$Inum_0_j + Inum_1_j \ge 5 \times Inuse_j . \quad (15)$$

4.5. Output Constraints for each GPC

Constraints (16)-(18) ensure that each of the 3 outputs of the GPC_i is connected to either an input of another GPC or an output of the compressor tree as long as the GPC_i is in use:

$$\sum_n q_{i,0,n} + \sum_{j \ne i} \sum_{kj} e_{i,j,0,kj} = Inuse_i \text{ , and} \quad (16)$$

$$\sum_n q_{i,1,n} + \sum_{j \ne i} \sum_{kj} e_{i,j,1,kj} = Inuse_i \text{ , and} \quad (17)$$

$$\sum_n q_{i,2,n} + \sum_{j \ne i} \sum_{kj} e_{i,j,2,kj} = Inuse_i . \quad (18)$$

4.6. Wiring Constraints

The constraints introduced in this section are necessary to ensure that the GPCs are wired together correctly so that the resulting circuit is actually a compressor tree. We simply ensure that the *rank*s of all inputs connected to each GPC are the same. The rank of a GPC input is either the rank of an input bit connected directly to the GPC, or the rank associated with a wire that connects one GPC to another. This, in turn, requires that we associate a rank with each GPC itself. First, we introduce some additional variables:

Rank(m) (Integer) – The rank of input bit m, which is provided as input to the ILP.

$Rank(p_{m,j,kj})$ (Integer) – The rank of input bit m with respect to its connection to GPC_j; only considered if $p_{m,j,kj} = 1$.

$Rank(e_{i,j,ki,kj})$ (Integer) – The rank of the wire $e_{i,j,ki,kj}$; only considered if $e_{i,j,ki,kj} = 1$.

$Rank_j$ (Integer) – The rank of GPC_j.

P_j – The set of wires from input bits to inputs of GPC_j:

$$P_j = \bigcup_{\substack{m=0 \\ kj \in \{0,1\}}}^{M} \left\{ p_{m,j,kj} \,\middle|\, p_{m,j,kj} = 1 \right\} \quad (19)$$

E_j – The set of wires connected directly to an input of GPC_j from another GPC, GPC_j:

$$E_j = \bigcup_{\substack{i \ne j \\ ki \in \{0,1,2\} \\ kj \in \{0,1\}}} \left\{ e_{i,j,ki,kj} \,\middle|\, e_{i,j,ki,kj} = 1 \right\} \quad (20)$$

Let $C_j = P_j \cup E_j$. The input constraint on GPC_j is satisfied if for each pair of distinct elements $c_1, c_2 \in C_j$, $rank(c_1) = rank(c_2)$. If

this constraint is satisfied, then without loss of generality, $Rank_j = rank(c_1)$. Here, we describe how to compute the rank of a GPC in terms of the connections to its inputs. The description uses induction on $Level_j$.

For the basis, let $Level_j = 0$. Then the only inputs connected directly to GPC_j are input bits, so E_j is empty. Consider some $p_{m,j,kj} \in P_j$. Then

$$rank(p_{m,j,kj}) = rank(m) - kj \; . \tag{21}$$

Recall that each GPC has inputs of ranks 0 and 1. The input constraint is satisfied if all input bits connected to rank 0 inputs of GPC_j have rank t, and all input bits connected to rank 1 inputs of GPC_j have rank $t+1$. If a rank t input was connected to an input of rank 1, then it would be overcounted; likewise, if a rank $t+1$ input was connected to an input of rank 0, it would be undercounted.

As an example, refer to Fig. 3. Input m_0, of rank 0, is connected to a rank-0 input of GPC_0 by the link $p_{0,0,0}$. Thus, $rank(p_{0,0,0}) = 0$. Likewise, input m_1, of rank 1, is connected to a rank-1 input of GPC_0 by the link $p_{1,0,1}$. Thus, $rank(p_{1,0,1}) = 1 - 1 = 0$. Note that the input constraint would not be satisfied if the respective ranks of the inputs of GPC_0 to which m_0 and m_1 were connected were reserved. Lastly, observe input bit m_2, of rank 2, which is connected to a rank-0 input of GPC_1. Then $Rank_1 = 2 - 0 = 2$.

For the induction hypothesis, suppose that we have computed $Rank_i$ for GPC_i such that $Level_i \leq L$, and that the input rank constraints are satisfied for GPC_i. Now, we describe how to compute the $Rank_j$ for GPC_j such that $Level_j = L+1$.

Without loss of generality, suppose that $e_{i,j,ki,kj} = 1$. $Rank_i$ is known by the induction hypothesis. As a counter GPC_i produces three outputs of rank $Rank_i$, $Rank_i + 1$, and $Rank_i + 2$ respectively. Thus the rank of the bit produced by the ki-th output of the GPC will be $Rank_i + ki$. Now, suppose this bit is connected to an input of rank kj of GPC_j, e.g. $e_{i,j,ki,kj} = 1$. Then to normalize the rank of $e_{i,j,ki,kj}$ with the other connections to inputs of GPC_i, we let:

$$rank(e_{i,j,ki,kj}) = Rank_i + ki - kj \; . \tag{22}$$

Eq. (21) can be used to compute the rank of input bits connected directly to GPC_j.

As an example, refer back to Fig. 3. From the above discussion, we know that $Rank_0 = 0$ and $Rank_1 = 2$. Consider edge $e_{0,2,1,0}$ which connects 2nd output of GPC_0 to a rank-0 input of GPC_2; with respect to Eq. (22), $ki = 1$ and $kj = 0$. Therefore $rank(e_{0,2,1,0}) = 0 + 1 - 0 = 1$. Next, consider edge $e_{1,2,0,1}$ which connects a rank-0 output of GPC_1 to a rank-1 input of GPC_2; with respect to Eq. (22), $ki = 0$ and $kj = 1$. Thus, $rank(e_{1,2,0,1}) = 2 + 0 - 1 = 1$. Therefore, the input rank constraint is satisfied for GPC_2 as well. Once again, note that a rank-1 wire ($e_{0,2,1,0}$) is connected to a rank-0 input of GPC_2, and that a rank-2 wire ($e_{1,2,0,1}$) is connected to a rank-1 input of GPC_2. Thus, GPC is correctly summing bits produced by other GPCs, in consecutive columns.

4.7. Further Improvements to the ILP

To speed up the ILP, we estimated the maximum number of logic levels required for each compressor tree, along with the maximum number of GPCs that would be needed at each level; the results of our previous heuristic [13] were used to provide these estimates. Furthermore, we added the constraint that the outputs of GPC on level L can only connect to inputs of GPCs on levels $L+1$ and $L+2$. This helps us to bound not only the number of GPCs, but also the number of $e_{i,j,ki,kj}$ variables in the ILP. This, in turn, reduces the size of the search space by eliminating variables. At the same time, there is no guarantee that a compressor tree restricted in this fashion will be globally optimal.

In many cases, the delay of the final adder may dominate the rest of the compressor tree. In some cases, the ILP reduces the arrival time of the least significant input bits to the final adder. To reduce the arrival time of these bits, the ILP is formulated so that more GPCs are allocated to earlier levels of the compressor tree and fewer GPCs are allocated to the later levels. This encourages the ILP to generate the least significant bits for the final adder in the earliest possible stages, which tends to reduce their arrival times.

5. Experimental Results

We compare the results of compressor tree synthesis produced by the ILP, solved using *Cplex*, against our greedy heuristic [13], and synthesis onto an adder tree. All the circuits were modeled by VHDL and synthesized by *Quartus-II* tool. We targeted the Altera Stratix-II [14]; we expect that similar results would be observed for the newer Stratix-III and also the Xilinx Virtex-5 FPGA; these architectures all support ternary addition and have 6-input LUTs.

We selected a set of benchmark circuits including multipliers (*m12x12*, *m16x16*), FIR filters (*fir3*) [1], motion estimation for video coding (*ME)* [3], and some circuits where multi-operand addition was exposed via circuit transformations [12]; only the compressor tree and final adder are synthesized here.

Fig. 4 shows the results of the experiments. Fig. 4(a) shows the area obtained by the 3 algorithms, while Fig. 4(b) shows the delay. In both charts, areas and delays are normalized to the results of the ILP. Fig. 4(a) shows that the ILP never produces a larger compressor tree than the heuristic. For all benchmarks, the area noticeably favors the ILP. In some cases, the area of the compressor tree produced by the ILP is less than the area of the adder tree; however, there are other cases—*Motion Est.* being the most dramatic—the adder tree consumes considerably less area. Significant improvements in area were observed for *G72x_2*, *mac*, *m12x12*, and *m16x16*, where the compressor trees produced by the heuristic consume more area than the adder tree, but the compressor trees produced by the ILP is less than the adder tree.

From Fig. 4(b), we can see that the delays of the compressor trees produced by both the ILP and the heuristic are significantly less than the delays of the adder tree. The ILP has the greatest relative advantage over the heuristic for *Motion Est.*, where the delay of the compressor tree produced by the heuristic is only slightly less than the delay of the adder tree, but the delay of the compressor tree produced by the ILP is significantly less than the delay of the compressor tree produced by the heuristic. On average, the area of the compressor trees produced by the heuristic are 10% larger than those of the adder tree, but the area of the ILP is 3% smaller. The delay of the compressor tree produced by the heuristic and the ILP were respectively 27% and 32% less than that of the adder tree.

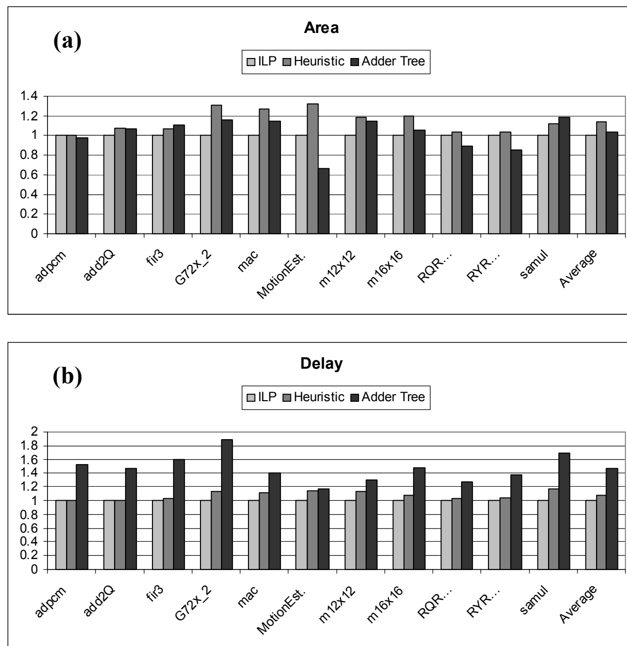

Figure 4. Area (a) and Delay (b) results for synthesis of compressor trees on the Stratix-II FPGA using an ILP formulation, the greedy heuristic [1], and adder trees. The results are normalized to the ILP.

In many cases, the number of logic levels produced by the ILP was the same as the number produced by the heuristic; the further reduction in delay was obtained since the arrival delay of the least significant inputs of the final adder were reduced. This causes that the carry propagation of the least significant bits in the final adder is performed in parallel with the preparation of the inputs to higher bits of the final adder. On the other hand, reduced ALM counts allow the LUTs comprising the compressor tree to be packed in more tightly. This reduces wire length, which in turn, reduces interconnect delays, which are quite pronounced in FPGAs.

The gains achieved by the ILP compared to the heuristic, shown in Fig. 4, come at a significant increase in runtime. The typical runtime for compressor tree generation using the heuristic was 2-4 seconds, while the runtime of the ILP was 70-200 seconds.

6. Conclusion

A new technique to map compressor trees onto FPGAs via Integer Linear Programming (ILP) has been presented. It was shown that the generalized parallel counters (GPCs) offer great flexibility for efficient synthesis of compressor trees onto FPGAs considering the LUT structures. Finding how to best select GPC configurations and wire a collection of GPCs together to build a compressor tree is quite complex. The problem was reformulated as ILP and several constraints were used to limit the search space. Compared to a prior greedy heuristic, the new technique produces compressor trees that are smaller (in terms of ALM count) and faster on Altera Stratix-II FPGAs. Although solving an ILP is NP-Complete in general, our formulation was able to find solutions in less than 4 minutes for all of our test cases.

References

[1] S. Mirzaei, A. Hosangadi, and R. Kastner, "High speed FIR filter implementation using add and shift method," *Int. Conf. Computer Design*, San Jose, CA, USA, Oct. 1-4, 2006.

[2] S. Sriram, K. Brown, R. Defosseux, F. Moerman, O. Paviot, V. Sundararajan, and A. Gatherer, "A 64 channel programmable receiver chip for 3G wireless infrastructure," *IEEE Custom Integrated Circuits Conf.*, San Jose, CA, USA, pp. 59-62, Sept. 18-21, 2005.

[3] C.Y. Chen, S-Y. Chien, Y-W. Huang, T-C. Chen, T-C. Wang, and L-G. Chen, "Analysis and architecture design of variable block-size motion estimation for H.264/AVC," *IEEE Trans. Circuits and Systems-I*, Vol. 53, No. 2, pp. 578-593, Feb., 2006.

[4] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, Vol. 34, pp. 349-356, May, 1965.

[5] O. Kwon, K. Nowka, and Jr. Swartzlander, "A 16-bit by 16-bit MAC design using fast 5:3 compressor cells," *Journal of VLSI Signal Procsesing*, Vol. 31, No. 2, pp. 77-89, June, 2002.

[6] H. Mora Mora, J. Mora Pascual, J. L. Sánchez Romero, and F. Pujol López, "Partial production reduction based on look-up tables," *Int. Conf. VLSI Design*, Hyderabad, India, pp. 399-404, January 3-7, 2006.

[7] J. Poldre and K. Tammemae, "Reconfigurable multiplier for Virtex FPGA family," *Int. Workshop on Field-Programmable Logic and Applications*, Glasgow, UK, pp. 359-364, Aug. 30 – Sept. 1, 1999.

[8] P. F. Stelling, C. U. Martel, V. J. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Trans. Computers*, Vol. 47, No. 3, pp. 273-285, March, 1998.

[9] J. Um and T. Kim, "Layout-aware synthesis of arithmetic circuits," *Design Automation Conf.*, pp. 207-212, June 10-14, 2002.

[10] A. K. Verma and P. Ienne, "Automatic synthesis of compressor trees: reevaluating large counters," *Design Automation and Test in Europe*, Nice, France, pp. 443-448, April 16-20, 2007.

[11] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, Vol. 13, pp. 14-17, Feb., 1964.

[12] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," *Int. Conf. Computer-Aided Design*, San Jose, CA, USA, pp. 791-798, Nov. 7-11, 2004.

[13] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient Synthesis of Compressor Trees on FPGAs," *Asia South Pacific Design Automation Conference (ASPDAC)*, January 2008.

[14] Altera, Corp. "The Stratix II device handbook, vol. 1 and 2," available online from http://www.altera.com/.

[15] Xilinx Corp. "Virtex-4 user guide," available online from http://www.xilinx.com.

[16] Altera, Corp. "Stratix II vs. Virtex-4 performance comparison," available online from http://www.altera.com/.