Efficient Synthesis of Compressor Trees on FPGAs

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 – FPGA performance is currently lacking for arithmetic circuits. Large sums of k > 2 integer values is a computationally intensive operation in applications such as digital signal and video processing. In ASIC design, compressor trees, such as Wallace and Dadda Trees, are used for parallel accumulation; however, the LUT structure and fast carry-chains employed by modern FPGAs favor trees of carry-propagate adders (CPAs), which are a poor choice for ASIC design. This paper presents the first method to successfully synthesize compressor trees on LUT-based FPGAs. In particular, we have found that generalized parallel counters (GPCs) map quite well to LUTs on FPGAs; a heuristic, presented within, constructs a compressor tree from a library of GPCs that can efficiently be implemented on the target FPGA. Compared to the ternary adder trees produced by commercial synthesis tools, our heuristc reduces the combinational delay by 27.5%, on average, within a tolerable average area increase of 5.7%.

I. Introduction

Despite more than 20 years of research on FPGAs, gaps in performance, power consumption, and area utilization between FPGAs and ASICs remain. These gaps are generally small for finite state machines and control-dominated circuits, but are pronounced for arithmetically intensive circuits. To improve the quality of arithmetic circuits synthesized on FPGAs, this paper describes a new method to synthesize compressor trees on FPGAs.

A compressor tree is a specific circuit implementation of *k*-input integer addition, generally under the assumption that $k > 2$. Wallace [19] and Dadda [7] introduced compressor trees more than 40 years ago, in the context of parallel multiplication. Compressor trees are also used in FIR filters [10], 3G wireless base station cards [13], and motion estimation in video coding algorithms such as H.264/ACV [5]. More generally, transformations can be applied to a general class of arithmetic circuits to form compressor trees by merging disparate addition operations with one another and with partial product reduction trees in parallel multipliers [18].

A *carry-propagate adder (CPA)* is a circuit that adds two integer values, A_0 and A_1, and produces the sum A_0+A_1. An *adder tree* is a circuit that adds *k*-values, $A_0, \ldots, A_{k-1}$, using a tree of CPAs. A compressor tree, in contrast, is a circuit constructed from *carry-save adders (CSAs)*, and produces two outputs, *S (sum)* and *C (carry)* such that $S+C = A_0 + \ldots + A_{k-1}$; a *CPA* is only required to add *S* and *C*. The superiority of compressor trees over adder trees for multi-input addition is well known [7, 19], and is beyond the scope of this paper.

High-performance FPGAs, however offer implementation of fast *ternary* carry-propagate addition by providing dedicated carry chains [1-3, 21]. Conventional wisdom has held that compressor trees cannot be synthesized efficiently onto FPGAs; this, in fact, is not the case.

The primary contribution of this paper is a demonstration of the fact that it is possible to effectively map compressor trees onto modern high performance FPGAs. Compressor trees synthesized using our approach were *27.5%* faster than ternary adder trees synthesized on an *Altera Stratix II* FPGA; on average, we suffered a tolerable area increase of *5.7%*.

The paper is organized as follows. Related work is summarized in Section II. Sections III and IV respectively introduce single-column counters and generalized parallel counters, which are basic building blocks used by our compressor tree synthesis method, which is described in Section V. Experimental results are then presented in Section VI, and lastly, Section VII concludes the paper.

II. Related Work

Over the years, numerous techniques in both hardware and software have been proposed to enhance FPGA performance. Most notably, carry chains [8] have been proposed to integrate efficient circuitry for arithmetic operations, such as shifting and carry-propagate addition into FPGA logic blocks; both *Xilinx* [20, 21] and *Altera* [2, 3] FPGAs feature such carry chains. With the *Stratix-II* architecture [2], *Altera* introduced a new logic architecture, which allows the 6-input LUTs (6-LUTs) to be decomposed into smaller LUTs that share several inputs. In particular, two 3-LUTs can be organized to compute the *carry* and *save* functions of a CSA, which acts as a 3:2 compressor; in conjunction with the carry chain, the logic block can be configured as a ternary CPA. Xilinx incorporated similar support for ternary addition into the *Virtex 5* logic blocks [21].

Another alternative for increased arithmetic acceleration is to integrate hard IP cores, such as DSP/MAC blocks, into FPGAs [22]. Although DSP cores are hand-optimized ASICs, Kuon and Rose [9] have argued that they do not offer much performance advantage for two reasons: (1) the cost of routing data to and from the blocks, whose position in the FPGA is fixed; and (2) mismatches in bitwidth (e.g. using a 9x9 bit multiplier to implement 5x5 bit multiplication). Thus, for operations whose bitwidth does not match that of the IP cores, the flexibility of LUT-based implementations is generally preferable.

Poldre and Tammemae [12] developed a software technique to synthesize compressor trees for parallel multipliers onto *Xilinx Virtex* FPGAs. Their technique was specific to the LUT structure and carry chains of that architecture. They reported delays that were 1.5x faster and used 1.28x less area than standard adder trees[1]. Our approach, in contrast, does not require or exploit carry chains, and is therefore amenable to any FPGA architecture.

To accelerate arithmetic circuits, Brisk et al. [4] introduced the *field programmable counter array (FPCA)*, a programmable array of parallel counters (see Section III), which could be integrated into an FPGA; the transformations of Verma and Ienne [18] are applied to a circuit in advance to expose large compressor trees that could be mapped onto the FPCA. The compressor tree synthesis heuristic

[1] The results are not shown in the paper, but are available online at http://www.pld.ttu.ee/~jp/FPL99/

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

presented here, in contrast, targets modern high-performance FPGAs that are available today: namely, the *Altera Stratix II/III* and the *Xilinx Virtex 4/5*, which employ the 6-LUTs as the core element of their logic cell.

The compressor tree synthesis heuristic presented in this paper is a form of technology mapping. Unlike traditional technology mappers, such as *FlowMap* [6] and its many descendants, our heuristic is limited in its domain to multi-input addition operation. To use a traditional technology mapper, a specific implementation of a compressor tree must be provided (e.g., Wallace Tree [19], Dadda Tree [7], *3*-greedy [14], etc.) to the mapper. Our approach, in contrast, simultaneously generates the specific implementation of the compressor tree and performs the mapping task at once; the remaining portions of the circuit are then synthesized using a traditional technology mapping algorithm.

III. Single Column Parallel Counters

Here, we describe different components that have been used in the past to construct compressor trees, typically for partial product reduction in parallel multipliers.

A *single column parallel counter* (also called an *m:n counter*) is a circuit that takes *m* input bits, counts the number of bits that are set to *1*, and outputs the result as an unsigned *n*-bit integer, a value in the range *[0, m]*. For a given *m*, the number of output bits, *n*, is:

$$n = \lceil log_2(m+1) \rceil \quad (1)$$

A *2:2* counter is typically called a *half-adder*; a *3:2* counter is called a *full-adder* (in the context of CPA design) and a *carry-save adder (CSA)* when used for parallel accumulation.

The use of CSAs for partial product reduction in parallel multipliers was introduced by in the 1960s [7, 19]; since then, numerous other approaches have been introduced as well. The logic-delay optimal (ignoring wiring delays) 3-greedy algorithm was introduced by Stelling et al. [14]; Um and Kim [16] extended this idea to account for wire delays and other issues related to layout. Verma and Ienne [17] developed an optimal *integer linear programming (ILP)* formulation of compressor tree synthesis from a library of counters ranging from *2* to *8* inputs; they also showed that an optimal compressor tree cannot be constructed purely from *m:n* counters.

All of the aforementioned compressor tree synthesis methods are tailored for ASICs; they do not perform well for FPGAs, and cannot compete with binary or ternary adder trees. Different components are necessary to successfully synthesize compressor trees on FPGAs.

IV. Generalized Parallel Counters

Let $B = (b_{k-1}b_{k-2}...b_0)$ be an *n*-bit positive binary integer. The *rank* of bit b_i is the subscript value *i*, which indicates its position in the integer. For example, in the binary value *0100*, the bit set to *1* has rank *2*. In general, bit b_i of rank *i* contributes a value of $b_i 2^i$ to the overall value represented by *B*. The following formula converts the bits in *B* to a decimal (base-10) value:

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

An *m:n* counter, by definition, always sums bits of the same rank. If the input bits all have rank *i*, then the output bits have ranks *i*, *i+1*, …, *i+(n-1)* respectively.

A *generalized parallel counter (GPC)* [15] is a counter that can sum bits having different ranks; all of the input bits of the same rank are referred to as a *column*. In principle, we could design a GPC that produces more than one output bit of each rank; to simplify our heuristic, however, we only consider GPCs that produce a single output bit of each rank. An *m:n* counter can implement a GPC by connecting all input bits of rank *i* to 2^i inputs of the counter. Of course, it is also possible to construct a GPC from basic gates as well; it turns out that *k*-input GPCs map quite well onto *k*-LUTs.

We assume that a GPC has at most *M* input bits and *N* output bits. Formally, a GPC is a tuple: $(K_{N-2}, K_{N-1}, ..., K_0; S)$, where the output *S* is an *N*-bit number. There are *N-1* columns, from rank *0* to *N-2*, with $K_i \geq 0$ input bits in the i^{th} column. For example, a *(1, 2, 3; 4)* GPC sums one rank-*2* input, two rank-*1* inputs, and three rank-*0* inputs, and produces four output bits. The largest possible output value that can be produced by this GPC is $1 \times 2^2 + 2 \times 2^1 + 3 \times 2^0 = 12$; $S = 4$ bits are required to express a value in the range *[0, 12]*. On the other hand, a *(1, 2, 3; 3)* GPC is infeasible, because *3* output bits cannot express a value in the range *[0, 12]*.

A legal GPC must satisfy the following two constraints:

$$\sum_{i=0}^{N-2} K_i \leq M \quad (3)$$

$$\sum_{i=0}^{N-2} K_i 2^i \leq 2^N - 1 \quad (4)$$

Constraint (3) ensures that the number of the bits in all columns does not exceed the maximum number of inputs, *M*. Constraint (4) ensures that the maximum value of the GPC (assuming all input bits are 1) does not exceed $2^N - 1$, the largest integer value that can be expressed with *N* bits. For a given *M* and *N*, there may be many different GPCs that satisfy these two constraints.

Fig. 1 provides an illustrative example. The input is two columns: three bits of rank-*1* and three bits of rank-*0*. Using two CSAs, as shown in Fig. 1(a), these bits can be reduced to three columns, one of which contains two bits. Thus, a CPA is required to sum the remaining bits; if this circuit was synthesized on an FPGA, it would require two layers of logic: the first layer for the two CSAs, and the second for the CPA. Fig. 1(b), in contrast, employs a *(3, 3; 4)* GPC, which reduces all six input bits to four columns, each of which contains a single bit; if mapped onto an FPGA, only one level of logic is required.

Fig. 2 shows a slightly larger example. The input is a set of columns, each having 5 bits. The compressor tree is generated using *(5, 5; 4)* GPCs. The layer of GPCs produces two rows of output bits, which can then be summed with a CSA.

An *M*-input GPC can be mapped efficiently onto *M*-LUTs, since one logic layer is required to compute the function. If the GPC has *N* outputs, then *N* *M*-LUTs are required to compute each of the output bits. Modern high-end FPGAs, such as the *Xilinx Virtex 4* and *5*, and *Altera Stratix II* and *III*, employ 6-LUTs as their basic logic elements. In practice, their logic cells are much more flexible: they contain multiple LUTs with shared inputs, carry chains, and sometimes local routing. The issue of shared LUT inputs can significantly affect the utilization of logic resources when mapping GPCs onto logic cells.

The *Altera Stratix II/III Adaptive Logic Module (ALM)* has two 6-LUTs with 6 shared inputs: thus, one ALM can compute any 6-input, 2-output Boolean function. This structure naturally lends itself to GPCs with $M = 6$ inputs and $N = 3$ or *4* outputs, respectively.

In Fig. 3(a), we map a *6*-input, *3*-output GPC onto two ALMs; one 6-LUT is unused, yielding a LUT utilization of 75%. In Fig. 3(b), we map a *6*-input, *4*-output GPC onto two ALMs, in this case, all 6-LUTs are used, so the utilization is 100%. Ideally, two *6*-input, *3*-output GPCs should be mapped onto the six *6*-LUTs contained in two ALMs; however, this is impossible because of shared LUT inputs, as shown in Fig. 3(c); four ALMs are necessary and the LUT utilization remains at 75%.

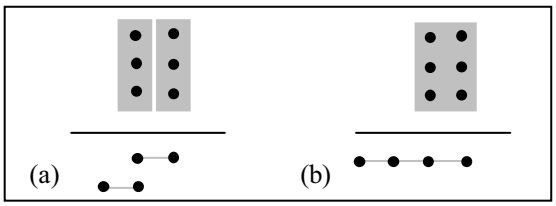

Fig. 1. Compressor tree mapping by (a) 3:2 counters (b) and a (3, 3; 4) GPC.

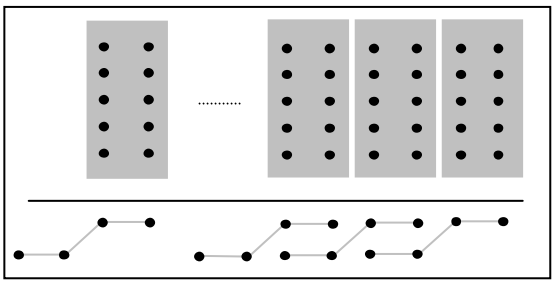

Fig. 2. Compressor tree synthesis using (5, 5; 4) GPCs

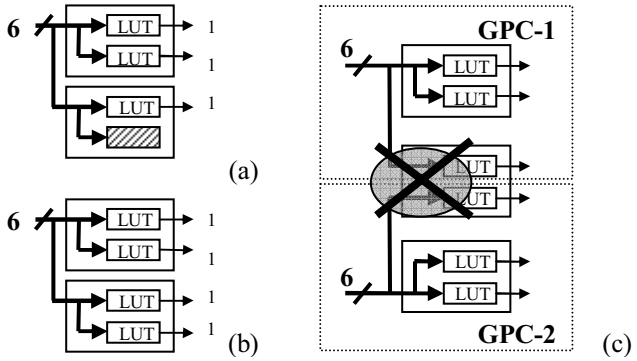

Fig. 3. An Altera ALM contains two 6-LUTs with shared inputs. A 6-input, 3-output GPC has 75% logic utilization (a); a 6-input, 4-output GPC has 100% logic utilization (b); two 3-output GPCs cannot be synthesized on 3 ALMs due to shared inputs (c).

V. Compressor Tree Synthesis Heuristic

This section describes a heuristic to construct a compressor tree from GPCs, which are then synthesized onto LUTs. The primary goal of the heuristic is to minimize the number of logic levels in the tree; minimizing the area (number of GPCs) is secondary. Like other work on more general forms of technology mapping [6], our heuristic cannot account for wire delays, because the placement and routing of the circuit is not known in advance.

A. Preliminary Definitions

A *dot* is an input/output bit of each GPC, represented visually, as in Figs. 1 and 2. This is typical of notation used in computer arithmetic textbooks, e.g. [11].

A *primitive GPC* is one that satisfies the I/O constraints. For example, if $M = 6$ and $N = 3$, there are *12* primitive GPCs, including *(1, 3; 3)* and *(2, 3; 3)*.

A *covering GPC* is one whose functionality, given I/O constraints, *cannot* be implemented by another GPC. For example, when $M = 6$ and $N = 3$, a *(2, 2; 3)* GPC is not a covering GPC, since it could be implemented with a *(2, 3; 3)* GPC with one rank-*0* bit set to *0*. For these I/O constraints, the set of covering GPCs is *{(0, 6; 3), (1, 5, 3), (2, 3; 3)}*. *(3, 0; 3)* is unreasonable because no bit of rank-*0* is summed.

The *compression ratio* of the GPC is the ratio of input to output bits. For example, the compression ratio of a *(3, 3; 4)* GPC is *6/4 = 1.5*; the compression ratio of a *(2, 3; 3)* GPC is *5/3 = 1.66*. GPCs with larger compression ratios are generally more effective at reducing the number of bits at each logic level in the compressor tree.

```
Synthesize_Compressor_Tree(Integer : M, N, k,
                           Array of Integers : Columns )
Step 1: Find_Covering_GPCs(M, N)
Step 2: Find_Primitive_GPCs(M, N)
Step 3: Order_Primitive_GPCs( )
Repeat {
   Step 4: Repeat {
              Col_Index = Max_Height_Column( Columns )
              Find_Next_GPC( Col_Index )
              Remove_Covered_Dots( )
           } Until all dots are covered or no reasonable
             GPC can be found
   Step 5: Connect_GPC_Inputs( )
   Step 6: Generate_Next_Stage_Dots( )
} Until k (or fewer) rows of dots remain
Step 7: Generate_Final_CPA( Columns )
```

Fig. 4. Compressor tree synthesis heuristic

When given a choice among several GPCs, our heuristic favors the one with the maximal compression ratio; given a choice between several GPCs with the same compression ratio, our heuristic favors the one that covers the largest number of dots.

We have identified two cases where the use of a particular GPC is *unreasonable*. In the first case, consider a *(3, 1; 3)* GPC. The one rank-*0* input dot will be propagated directly to the rank-*0* output; thus, it can be routed directly, bypassing the GPC. In the second case, a GPC may fail to reduce the number of dots: a *(1, 2; 3)* GPC takes 3 input dots, representing a value in the range *[0, 4]*; therefore, it produces 3 output dots; in general a GPC that fails to reduce the number of dots is not useful.

B. Heuristic

This section introduces the heuristic for compressor tree synthesis, using the definitions introduced in the previous section. All else being equal, the heuristic favors GPCs with higher compression ratios. This approach tends to favor both of our objectives: it reduces the number of logic levels in the compressor tree and the number of GPCs used.

Fig. 4 shows pseudocode for the mapping heuristic, which contains seven main steps. The inputs to the algorithm are:

M – the input constraint of the GPC (typically the LUT size of the target FPGA, e.g., *6* for *Xilinx Virtex 4*/*5* and *Altera Stratix II/III*.

N – the output constraint of the GPC.

k – the number of input rows of the final adder (typically 2, but *3* in the case of the ternary adders in the *Stratix II/III* and *Virtex 5*).

Columns – the set of bits to be summed; in principle, an array of non-negative integers, where the i^{th} integer in the array is the number of bits in the rank-*(i-1)* column.

The output is a compressor tree constructed from GPCs that satisfy

the I/O constraints; the mapping does not include the final adder, but this portion of the mapping is trivial. Our experiments are performed on the *Stratix II* FPGA by *Altera*. In this case, $M = 6$, $N = 4$, and $k = 3$.

Here, we summarize the seven main steps in the mapping heuristic. The first three steps create an appropriate library of GPCs, and can be performed once for each pair of values M and N; the GPC library can then be stored in a database offline. The remaining steps perform the actual mapping.

Step 1. The first step is to generate the set of covering GPCs, given M and N. This step is straightforward and deterministic, and converges quickly for small constant values of M and N.

Step 2. The second step is to generate the set of primitive GPCs, given M and N. In the general-case, the size of this set may be exponential in M and/or N; however, M and N are fairly small constants (*6* and *4*, in our case). The compressor tree itself is constructed from primitive GPCs, but in the final netlist each primitive GPC is represented by its corresponding covering GPC with some inputs set to 0.

Step 3. The third step of the heuristic is to sort the primitive GPCs according to their compression ratio. All primitive GPCs with the same compression ratio are then locally sorted by the number of covering dots. The resulting set is used to cover the adder tree in an iterative fashion during steps 4-6.

Steps 4-6 occur inside a loop that generates the compressor tree. Each execution of these three steps creates another logic layer in the tree, which reduces the number of rows of bits to be summed. These steps repeat until there are at most k rows, e.g., at most k bits per column. At this point, the heuristic terminates, and a k-input CPA is created to sum the remaining rows.

Step 4. The fourth step covers all of the columns of the current logic level of the compressor tree with GPCs. Connections between the current and preceding logic levels are then created in Step 5, and Step 6 generates the new set of columns for the following logic layer.

First, a column containing the maximal number of dots is selected as the *base column*. This tends to favor GPCs with high compression ratios. To find the best GPC for the base column, the ordered set of GPC primitives is search from highest to lowest priority.

Suppose that the i^{th} column is selected as the base column. Since GPCs can compress bits in multiple columns, we can, for example, consider the dots in columns *i+1*, *i+2*, …, *i+(N-1)* in conjunction with the dots in column *i*: we call this a *forward search*, as illustrated in Fig. 5(a); likewise, we could also consider the dots column *i* in conjunction with dots in columns *i-1*, *i-2*, …, *i-(N-1)*: we call this a *backward search*, as illustrated in Fig. 5(b).

The first GPC that fits the base column and its following columns (forward search) is chosen as GPC_F; likewise, the first GPC that fits the base column and its preceding columns (backward search) is chosen as GPC_B; between GPC_F and GPC_B, the one of higher priority is selected. In Fig. 5(a), the forward search finds $GPC_F = (3, 5; 4)$, and in Fig. 5(b), the backward search finds $GBC_B = (5, 2; 4)$. The respective compression ratios are $CR_F = 8/4 = 2$ and $CR_B = 7/4 = 1.75$. Since $CR_F > CR_B$, GPC_F has a higher priority and is selected.

The reason to do both a forward and a backward search is that the input dots may have an asymmetric distribution. This can occur in FIR and other types of filters. For a symmetric distribution—for example, adding several *32*-bit integers—a greedy forward search, starting from the least significant column toward the most significant column, would probably suffice.

Step 4 repeats until all of the dots in the current stage of the compressor tree are covered, or no primitive GPC can be found for any of the remaining dots in the current stage. Any uncovered dots are propagated to the next stage in Step 6.

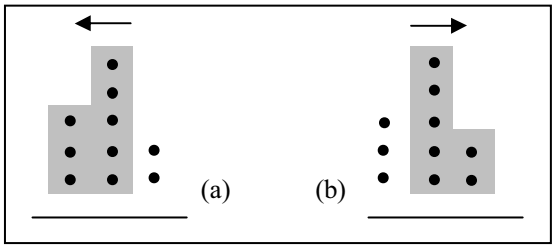

Fig. 5. Forward (a) and backward (b) search

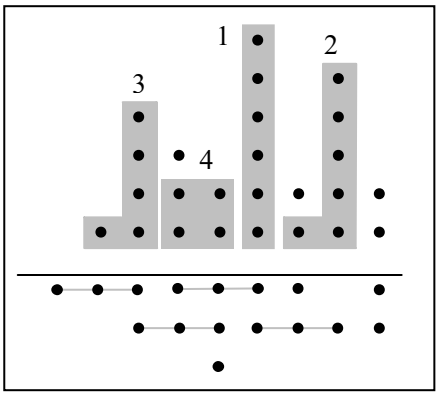

Fig. 6. Example illustrating four iterations of Step 4 on an asymmetric set of input columns ($M = 6$; $N = 3$).

Fig. 6 shows an example of an asymmetric set of input bits. Five iterations of Step 4 are required to cover (most) of the bits by GPCs. In this example, $M = 6$ and $N = 3$, so each GPC can consume at most *6* input bits and can produce at most *3* output bits.

1. The tallest column has six bits, so a *(0, 6; 3)* GPC, labeled *'1'* in Fig. 6, covers all of the bits in the column.
2. The second-tallest column has five bits; by using a *(1, 5; 3)* GPC, labeled *'2'* in Fig. 6, an extra bit from adjacent column on the left (higher rank) is covered as well.
3. The third-tallest column has four bits; by using a *(1, 4; 3)* GPC, labeled *'3'* in Fig. 6. The one bit from the column to the left (higher rank) is covered as well.
4. The fourth-tallest column has three bits. One possibility would be to use a *(3, 1; 3)* GPC to cover these three bits, along with an extra bit in the preceding column; however, this GPC is unreasonable because the rank-*0* input bit is simply propagated directly to the rank-*0* output. Therefore, we use a *(2, 2; 3)* GPC, labeled *'4'*, in Fig. 6, instead. We cannot use a GPC that covers all 5 bits in these two columns, because its maximum value would be *8*, which requires 4 output bits.
5. Four bits are left unmapped. There are only two rank-*0* bits; they do not need to be added until the final CPA. The rank-*2* and rank-*4* bit could be covered by a *(1, 0, 1; 3)* GPC; however, this counter is unreasonable because it produces more output bits than it consumes, and the rank-*0* and rank-*1* outputs are equal to the rank-*0* and rank-*1* inputs respectively. Thus, all four of these bits are propagated to the next level of the compressor tree as inputs in their respective columns.

Step 5. In the fifth step of the mapping heuristic, the GPCs that were created in Step 4 are connected to the GPCs in the previous stage (or the compressor tree inputs, if appropriate). Each GPC output in stage *i-1* is connected to a GPC input in stage *i*, such that the respective GPC input and output in question have the same rank.

Step 6. In the sixth step of the mapping heuristic, the columns for the next layer of logic are generated from the GPCs that are allocated in

Step 4. Any uncovered dots are propagated into the appropriate columns. A new layer of logic is generated, comprised either of GPCs (return to Step 4) or a *k*-ary CPA (Step 7, followed by termination).

Step 7. If there are no more than *k* rows of bits remaining, then there is no need for further GPCs; instead, these bits are summed using a *k*-ary CPA, which produces one final row of output bits. Once the CPA is generated, the heuristic terminates.

VI. Experimental Results

A. Overview

The algorithm described in the preceding section was implemented and tested in a synthesis flow targeting the *Altera Stratix II FPGA*. We synthesized a set of arithmetic circuits used by Verma and Ienne [18] to demonstrate the effectiveness of a flowgraph rewriting heuristic to expose compressor trees; we also synthesized a FIR filter, a 12x12 parallel multiplier, and an internally-developed 1-dimensional systolic array implementation of the variable block size motion estimation phase of H.264/AVC *(ME)*. We isolated the multi-input addition operations, and synthesized each one as a combinational circuit, i.e., without pipelining it. We synthesized each multi-input addition operation using three different approaches:

GPC: Compressor tree synthesis using GPCs, as described in the preceding section, with $M = 6$ and $N = 4$. The compressor tree produced 3 outputs summed by ALMs configured as a ternary CPA.

ADD: Synthesis on ALMs configured as a tree of ternary CPAs. This is considered to be the state-of-the-art at present because it exploits fast carry-chains.

3GD: Compressor tree synthesis using the 3-greedy algorithm of Stelling et al. [14]. Each compressor tree was constructed from ALMs configured as CSAs, rather than CPAs; as such, it cannot exploit carry chains. For ASICs, 3GD is delay-optimal under the assumption of zero wire delay. This assumption is fairly unrealistic, both for ASICs (where wires do not scale as well as transistors) and FPGAs, where wire delay often dominates logic delay for large combinational circuits. The inclusion of 3GD illustrates the fact that CSAs are the incorrect building blocks for compressor tree synthesis on FPGAs.

Our compressor/adder tree synthesis software produced structural VHDL that was mapped onto the FPGA using *Synplicity's Simplify Pro*, and placed-and-routed with *Altera*'s *Quartus-II* software.

In the case of *3ADD*, extra care had to be taken when generating VHDL for the ternary adder tree. The Quartus-II fitter does not automatically map multi-input addition operations to compressor tree. For example, a straighforward product accumulation statement, of the form $S <= i0 + i1 + ... + ik$ is mapped to a mixture of binary adders and LUTs, rather than ternary adders. On the other hand, a statement of the form $T = i0 + i1 + i2;\ S <= T + i3 + i4$ is mapped directly to ternary adders, and the result is a faster implementation; we generated VHDL using the latter coding style.

B. Results

The delay (ns) of the compressor trees synthesized using *GPC*, *3ADD*, and *3GD* is shown in Figs. 7. In all cases but one, GPC achieved a smaller delay than either *3ADD* or *3GD*; however, in the case of *G721_X*, *GPC* and *3ADD* found the solution with the same delay. In this case, four 32-bit integers are summed; thus, two layers of logic are required for both the adder and the compressor tree. On average, the combinational delay of GPC was *27.5%* less than *3ADD* and *28.6%* less than *3GD*.

Fig. 8 shows the area (ALMs) of the three synthesis methods. On average, *GPC* used *74.1* ALMs, *3ADD* used *70.1*, and *3GD* used *112.8*. In the case of *SAMUL* and *FIR3*, *GPC* required fewer ALMs than *3ADD*, which was due to the asymmetric input patterns in the latter case. In a few cases, the difference in area between *GPC* and *3ADD* was negligible; however, the overall trend for area favored *3ADD*.

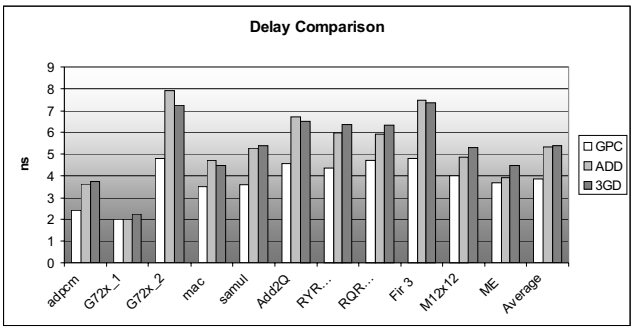

Fig. 7. Delay (ns) from mapping multi-input addition operations onto the Stratix-II FPGA.

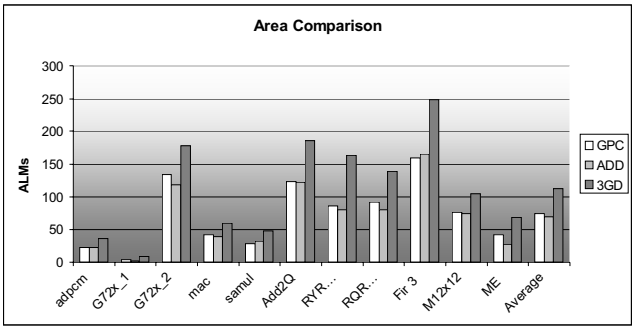

Fig. 8. Area (ALMs) from mapping multi-input addition operations onto the Stratix-II FPGA.

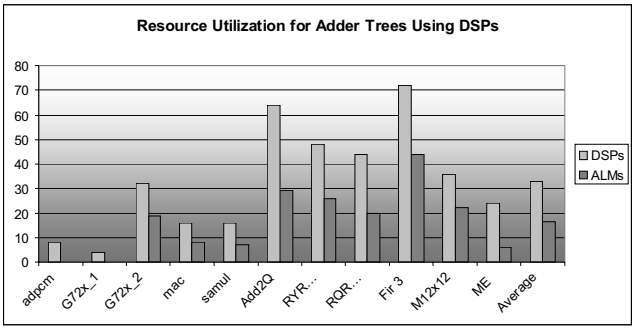

Fig. 9. Resources (DSPs and ALMs) used when by mapping multi-input additions onto adder trees using DSP blocks.

Modern high-performance FPGAs contain DSP/MAC blocks for arithmetic acceleration; however, these blocks are optimized for high-throughput, high-frequency pipelined accumulation, rather than parallel accumulation. Nonetheless, we tried to synthesize ternary adder trees using them. The delay using this approach was on average, more than twice the delay of *3GD*; Fig. 9 shows the resource usage.

On average, this approach used *33.1* 9-bit DSP blocks and *16.5* ALMs, compared to *70.1* ALMs, on average, for *3ADD*. A comparison of *49.6* "resources" to *70.1* ALMs is spurious, because a DSP block is considerably more complex than an ALM. Due to the high delay of this last approach, we do not advocate using an adder tree built from DSP blocks unless resource constraints prevent the

use of *GPC* or *3ADD*. At the same time, a larger FPGA device from the same family is likely to be a preferable choice than an adder tree with unreasonably high delay.

These experiments focused specifically on the ALM structure of *Altera Stratix II* FPGAs; the *Stratix III* uses the same ALM architecture; likewise, we would expect to observe comparable results for the *Xilinx Virtex 5*, which also supports ternary addition.

For prior FPGAs, that used *3*- or *4*-LUTs, it is not clear whether the proposed approach would be beneficial. Compressor trees would require more layers of logic; however, on the other hand, ternary addition is not supported, and carry chain technology was not advanced as it is today.

VII. Conclusion and Future Work

A novel technique to map compressor trees onto FPGAs has been proposed. In our experiments with the *Altera Stratix II* FPGA, our technique reduced the critical path delay by 27.5%, on average, compared to ternary adder trees, with a tolerable average increase in ALM usage of 5.71%.The mapping heuristic used GPCs, rather than CSAs and single column parallel counters, which are generally used in ASIC synthesis of compressor trees.

Conventionally, it has been thought that compressor trees do not map well onto FPGAs; we have shown that this is not the case. More precisely, however, these assumptions are still shown to hold true for compressor trees synthesized from CSAs, not necessarily compressor trees synthesized from larger *m:n* counters and *GPCs*. It is possible that other common components for arithmetic circuit design, such as *6:2 compressors* may also be useful in compressor tree synthesis for FPGAs; we leave this investigation open for future work.

It is not immediately clear whether GPCs are useful in compressor tree synthesis for ASICs. On the one hand, GPCs can be constructed from smaller components, such as CSAs, and *m:n* counters; thus, any technique that constructs compressor trees from CSAs and *m:n* counters implicitly has the ability to use CSAs. On the other hand, Verma and Ienne [17] have shown that better architectures for *m:n* counters occur when they are constructed from basic gates, rather than CSAs and smaller *m:n* counters; thus, it is possible that the same property holds for GPCs as well; in this case, it is possible, although not guaranteed, that GPCs could be useful components in ASIC synthesis of compressor trees.

References

[1] Altera, Corp. “Stratix-II vs. Virtex-4 performance comparison, ver. 2.0”, September, 2006, available online from http://www.altera.com/

[2] Altera, Corp. “The Stratix-II device handbook” available online from http://www.altera.com/

[3] Altera, Corp. “The Stratix-III device handbook,” available online from http://www.altera.com/

[4] P. Brisk, A. K. Verma, H. Parandeh-Afshar, and P. Ienne, “Enhancing FPGA performance for arithmetic circuits,” *Design Automation Conf.*, San Diego, CA, USA, June 4-8, 2007.

[5] 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.

[6] J. Cong and Y. Ding, “FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” *IEEE Trans. CAD*, Vol. 13, No. 1, pp. 1-12, Jan., 1994.

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

[8] S. Hauck, M. M. Hosler, and T. W. Fry, “High-performance carry chains for FPGAs,” *IEEE Trans. VLSI Systems*, Vol 8, No. 2, pp. 138-147, April, 2000.

[9] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *IEEE Trans. CAD*, Vol. 26, No. 2, pp. 203-215, Feb., 2007.

[10] 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.

[11] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*. New York: Oxford Press, 2000.

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

[13] 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.

[14] 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.

[15] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, “A compact high-speed parallel multiplication scheme,” *IEEE Trans. Computers*, Vol. C-26, No. 10, pp. 948-957, October, 1977.

[16] J. Um and T. Kim, “Layout-aware synthesis of arithmetic circuits,” *Design Automation Conf.*, New Orleans, LA, USA, pp. 207-212, June 10-14, 2002.

[17] A. K. Verma and P. Ienne, “Automatic synthesis of compressor trees: reevaluating large counters,” *Design Automation and Test in Europe*, pp. 1-6, Nice, France, April 16-20, 2007.

[18] 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.

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

[20] Xilinx Corporation, *Virtex-4 User Guide*, available online from: http://www.xilinx.com/

[21] Xilinx Corporation, *Virtex-5 User Guide*, available online from: http://www.xilinx.com/

[22] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, “A hybrid ASIC and FPGA architecture,” *Int. Conf. Computer-Aided Design*, San Jose, CA, USA pp. 187-194, Nov. 10-14, 2002.