Field Programmable Compressor Trees: Acceleration of Multi-Input Addition on FPGAs

ALESSANDRO CEVRERO, PANAGIOTIS ATHANASOPOULOS,
HADI PARANDEH-AFSHAR, and AJAY K. VERMA
Ecole Polytechnique Fédérale de Lausanne (EPFL)
HOSEIN SEYED ATTARZADEH NIAKI
Royal Institute of Technology, Sweden
CHRYSOSTOMOS NICOPOULOS
University of Cyprus
FRANK K. GURKAYNAK
Swiss Federal Institute of Technology, Zurich (ETHZ)
and
PHILIP BRISK, YUSUF LEBLEBICI, and PAOLO IENNE
Ecole Polytechnique Fédérale de Lausanne (EPFL)

Multi-input addition occurs in a variety of arithmetically intensive signal processing applications. The DSP blocks embedded in high-performance FPGAs perform fixed bitwidth parallel multiplication and Multiply-ACcumulate (MAC) operations. In theory, the compressor trees contained within the multipliers could implement multi-input addition; however, they are not exposed to the programmer. To improve FPGA performance for these applications, this article introduces the Field Programmable Compressor Tree (FPCT) as an alternative to the DSP blocks. By providing just a compressor tree, the FPCT can perform multi-input addition along with parallel multiplication and MAC in conjunction with a small amount of FPGA general logic. Furthermore, the user can configure the FPCT to precisely match the bitwidths of the operands being summed. Although an FPCT cannot beat the performance of a well-designed ASIC compressor tree of fixed bitwidth, for example, 9×9 and 18×18-bit multipliers/MACs in DSP blocks, its configurable bitwidth and

Authors' addresses: A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. K. Verma, P. Brisk, Y. Leblebici, and P. Ienne, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland; email: Philip.brisk@gmail.com; C. Nicopoulos, University of Cyprus, Nicosia; H. S. A. Niaki, Royal Institute of Technology, Sweden; F. K. Gurkaynak, Swiss Federal Institute of Technology, Zunich (ETHZ), Zunich, Switzerland.

ability to perform multi-input addition is ideal for reconfigurable devices that are used across a variety of applications.

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 Programmable Gate Array (FPGA), compressor tree, Field Programmable Compressor Tree (FPCT)

ACM Reference Format:
Cevrero, A., Athanasopoulos, P., Parandeh-Afshar, H., Verma, A. K., Niaki, H. S. A., Nicopoulos, C., Gurkaynak, F. K., Brisk, P., Leblebici, Y., and Ienne, P. 2009. Field programmable compressor trees: Acceleration of multi-input addition on FPGAs. ACM Trans. Reconfig. Technol. Syst. 2, 2, Article 13 (June 2009), 36 pages. DOI = 10.1145/1534916.1534923. http://doi.acm.org/10.1145/1534916.1534923.

1. INTRODUCTION

Multi-input addition occurs in many industrially relevant applications, such as FIR filters [Mirzaei et al. 2006], the *Sum-of-Absolute Difference (SAD)* computation used in video coding [Chen et al. 2006], and correlators used in 3G wireless base station channel cards [Sriram et al. 2005], among others. Verma et al. [2008] introduced a generally applicable set of data flow graph transformations to form *compressor trees* by merging disparate addition operations together and with the partial product reduction trees of multipliers used in the same computations. These transformations, however, targeted an ASIC design flow and are not well suited to FPGAs for two reasons. First, FPGA logic cells are not designed to accelerate carry-save addition. Second, the DSP blocks that are embedded within modern FPGAs cannot perform multi-input addition and cannot realize multipliers whose partial product reduction trees have been merged with other addition operations. To improve FPGA performance for arithmetic circuits, this article introduces the *Field Programmable Compressor Tree (FPCT)*, a programmable accelerator for carry-save arithmetic, as an alternative to the DSP blocks that are employed today.

DSP blocks are efficient ASIC integer multiplications, with additional features such as *Multiply-ACcumulate (MAC)*. The partial product reduction trees inside the multipliers are compressor trees built from carry-save adders, such as, Wallace [1964] and Dadda [1965] trees; however, these compressor trees are not directly accessible to the programmer, and cannot perform multi-input integer addition. Instead, multi-input addition must be realized on the general logic of the FPGA, which is comprised of programmable *look-up tables (LUTs)* with bypassable flip-flops to facilitate sequential circuits. In addition to LUTs, logic cells contain carry chains, that is, fast *Carry Propagate Adders (CPAs)*, to improve arithmetic performance. Although *adder trees*, namely, trees built from *Carry Propagate Adders (CPAs)* can perform multi-input addition, carry-save-based compressor trees are far superior in ASIC technology. Although it is possible to synthesize compressor trees directly on the general logic of an FPGA [Parandeh-Afshar et al. 2008b; 2008c], connections from one level of

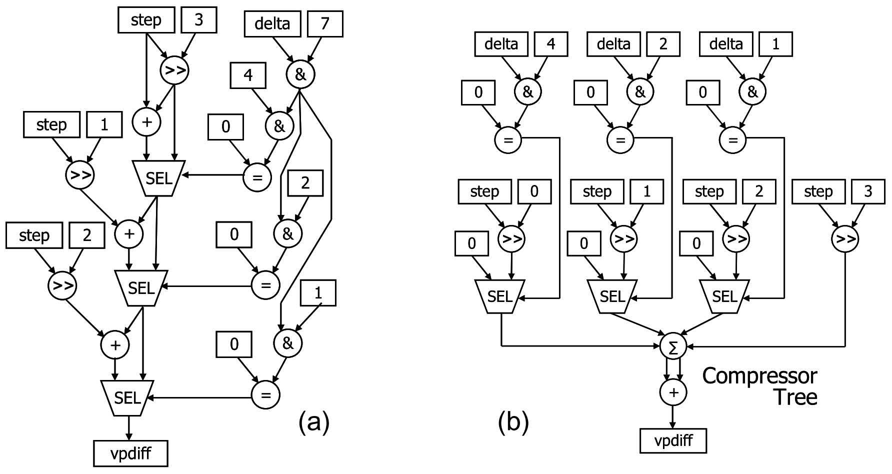


Fig. 1. A kernel of the adpcm benchmark, originally written in C [Lee et al. 1997]: (a) a dataflow graph of the circuit is shown following if-conversion [Allen et al. 1983]; and (b) rewritten to merge three addition operations into a compressor tree [Verma et al. 2008].

the compressor tree to the next must use the programmable routing network, which entails considerable overhead.

The FPCT, in contrast, employs a limited programmable interconnect that permits the user to adapt it to match the bitwidth and structure of the specific multi-input addition and/or multiplication operation to be synthesized upon it. This imbues the FPCT with greater flexibility than the DSP block, and with a leaner interconnect compared to the FPGA general logic. On the other hand, applications whose operations and bitwidths precisely match the DSP blocks will not benefit from the migration to an FPGA containing FPCTs, unless the transformations of Verma et al. [2008] can be applied. To satisfy both classes of users (i.e., those for whom the DSP blocks are too limited and those for whom the DSP blocks offer precisely the desired functionality) we recommend that FPGAs contain a mixture of FPCTs and DSP blocks. That being said, the primary focus of this article is the design and evaluation of the FPCT, including direct comparisons with DSP blocks. Determining the ideal ratio of FPCTs to DSP blocks within a larger FPGA is left open for future work.

As a motivating example, Figure 1 shows a *Data Flow Graph (DFG)* taken from the *adpcm* benchmark [Lee et al. 1997]. Figure 1(a) shows a software implementation after applying if-conversion [Allen et al. 1983], a compiler optimization that increases parallelism. Given a statement of the form *if(condition)* $x = f_1(\ldots)$ *else* $x = f_2(\ldots)$, which executes sequentially in software, if-conversion, in contrast, computes f_1 and f_2 in parallel, and selects the correct result using a multiplexor; the selection bit of the multiplexor is the *condition* variable. This optimization has the advantage of overlapping the computation of f_1 and f_2 with the resolution of *condition*, but typically increases area.

The critical path of the circuit contains three adders separated by multiplexors (“SEL”ect operations). Figure 1(b) shows the circuit after applying the transformations of Verma et al. [2008], which merge the three CPAs into a single 4-input addition operation, which can be implemented using a compressor tree. Figure 1(b), in fact, is a parallel multiplier built using an irregular partial product generator, which prevents the synthesis of the transformed circuit onto a DSP block. Instead, the general logic of the FPGA must be used, or, alternatively, an FPCT.

In an ASIC synthesis flow, the transformations reduce the critical path delay by 46% [Verma et al. 2008]. In an FPGA environment, however, these transformations are less effective. Using Parandeh-Afshar et al.’s [2008b] compressor tree synthesis heuristic, synthesizing the DFG shown in Figure 1(b) on an Altera Stratix II FPGA reduced the critical path delay by 33% compared to synthesizing the DFG in Figure 1(a). If the FPGA contained an FPCT, in contrast, we estimate that the critical path reduction would be 49%, comparable to the gains achieved using an ASIC synthesis flow. We estimate that the area of the FPCT is approximately equal to that of ten *Adaptive Logic Modules (ALMs)*, Altera’s logic cells.

Synthesizing the untransformed circuit in Figure 1(a) on the FPGA general logic requires 41 ALMs; in contrast, synthesizing the transformed circuit in Figure 1(b) requires 20 ALMs and one FPCT, an estimated area reduction of 25%.

To summarize, the DSP block could not be used for either circuit in Figure 1(a) or 1(b). Using the FPCT for the compressor tree in Figure 1(b) improved both delay and area compared to synthesis on the general logic of an FPGA.

2. ARITHMETIC PRIMITIVES

2.1 Compressor Trees and the FPCT

A *Carry-Propagate Adder (CPA)* is a circuit that computes the sum of two integers. Examples of CPAs include ripple-carry and carry-select adders, among others. Now, suppose that we want to add n integers, $A_0 + \ldots + A_{n-1}$, in parallel. One possibility is to use an *adder tree*, namely, a tree of CPAs. An alternative is a *compressor tree,* built from carry-save adders, that produces a *sum (S)* and *carry (C)*, where $S + C = A_0 + \ldots + A_{n-1}$. A CPA is only required to compute the final sum, $S + C$. The advantage of compressor trees over adder trees is that the output bits of the compressor tree have nonuniform arrival times at the final CPA. Fast CPAs, such as carry-select adders, can exploit the nonuniform arrival times to reduce the critical path delay of the circuit; slower CPAs, such as ripple-carry adders, cannot, and are not used.

Figure 2 illustrates the uses of compressor trees explored in this article: multi-input addition (Figure 2(a)); parallel multiplication (Figure 2(b)); a DSP block that performs parallel multiplication and MAC (Figure 2(c)); and the FPCT (Figure 2(d)). The DSP block cannot perform multi-input addition because its compressor tree is not directly accessible without first going through

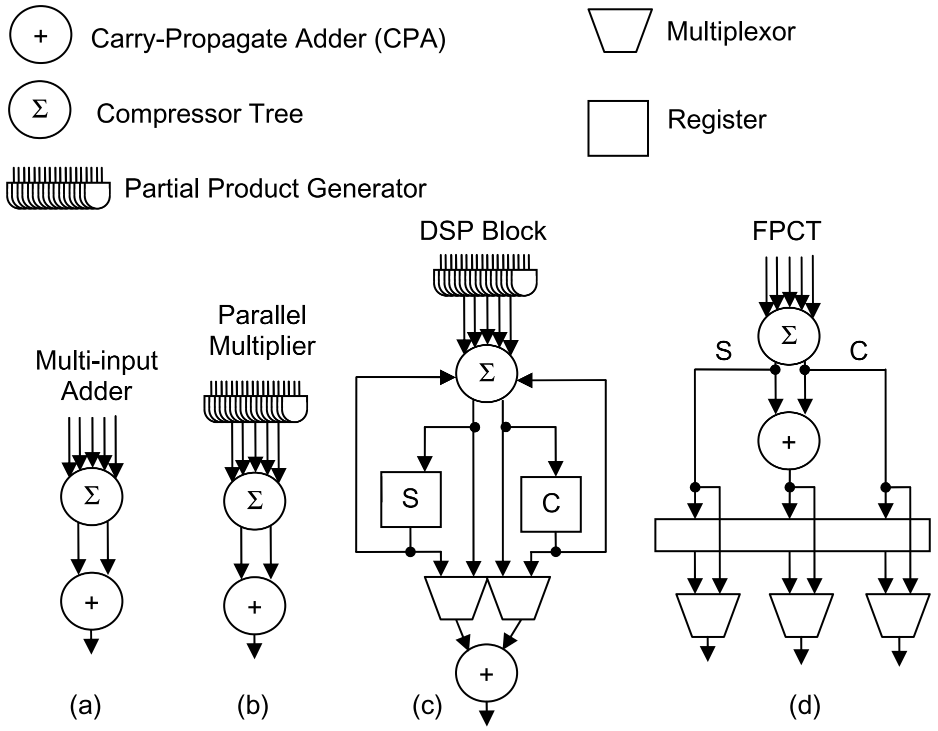


Fig. 2. (a) A compressor tree used for multi-input addition; (b) a compressor tree used for parallel multiplication; (c) a DSP block contains a compressor tree that is used for parallel multiplication and MAC operations; and (d) the FPCT; unlike (a)–(c), the user can program the bitwidth of the FPCT.

the *Partial Product Generator (PPG)*. The FPCT remedies the situation, as it does not employ a PPG; instead, the PPG is synthesized on the FPGA general logic. To compute a MAC, the S and C registers, which are part of the FPCT, are connected to the compressor tree inputs via the FPGA's programmable routing network. To summarize, in conjunction with a small amount of general FPGA logic, the FPCT can be configured to offer the same functionality as the DSP block.

2.2 Dot Notation

Let $A = a_{n-1}a_{n-2}\ldots a_0$ be a n-bit binary integer; a_0 is the least significant bit and a_{n-1} is the most significant bit. The subscript i of a_i is called the *rank* of a_i. a_i contributes $a_i 2^i$ to the total value of A. Multi-input addition can be represented using *dot notation*. Each n-bit integer A_i is represented by a row of dots, where the rightmost dot represents the least significant bit. A dot can be eliminated when it is known (due to external information, such as constant values) that the value of the corresponding bit is always 0. A *column* is the set of dots having the same rank. The addition of k n-bit integers is represented as a rectangular pattern of dots with k rows, n columns.

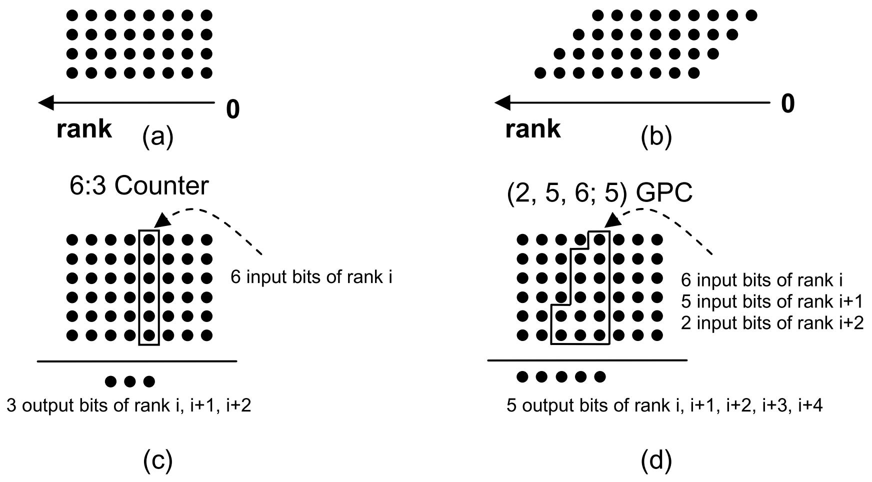


Fig. 3. Examples of dot notation: (a) addition of four 8-bit integers; (b) 4x8-bit multiplication (after partial product generation); (c) a 6:3 counter; (d) a (2, 5, 6; 5) GPC.

Figure 3(a) shows an example for $k = 4$ and $n = 8$. The partial product generator for an $m \times n$-bit multiplier produces a trapezoidal pattern of dots, as shown in Figure 3(b). When constants are added and/or multiplied, the pattern of dots can become irregular; this occurs, for example, in FIR filters [Mirzaei et al. 2006], which are often built from highly specialized constant multipliers.

2.3 Parallel Counters

An $m : n$ *counter* is a circuit that takes m input bits, counts the number of input bits that are set to 1, and outputs this quantity as an n-bit unsigned binary integer: a value in the range $[0, m]$. The number of output bits, n, is computed by the following formula.

$$n = \lceil log_2 (m + 1) \rceil \tag{1}$$

A *Generalized Parallel Counter (GPC)* [Stenzel et al. 1977] counts input bits from several adjacent columns. Formally, an m-input, $n-$output GPC is specified by a tuple: $(m_{k-1}, m_{k-2}, \ldots, m_0; n)$, where m_i is the number of input bits of rank i. The maximum value that can be expressed by the GPC is $M = m_0 2^0 + m_1 2^1 + \ldots + m_{k-1} 2^{k-1}$; an $m : n$ counter is a degenerate GPC in which all bits have the same rank. A large $m : n$ counter can also implement a GPC: Each input bit of rank i is connected to 2^i inputs of the $m : n$ counter. Let i be the minimum rank of a GPC input; then, the outputs have ranks i, $i + 1$, $\ldots$, $i + n - 1$, as illustrated by Figure 3(c) for a 6:3 counter and Figure 3(d) for a (2, 5, 6; 5) GPC.

3. RELATED WORK

Early FPGAs used k-input LUTS (k-LUTs) as their logic cells. k-LUTs are universal in the sense that any Boolean logic function can be implemented using only k-LUTs. They are good building blocks for state machines and control-dominated circuits, but are inefficient for arithmetic circuits. For this reason, commercial FPGA vendors now embed DSP blocks [Zuchowski et al. 2002], multipliers, and memories into their devices.

3.1 DSP Blocks

DSP blocks are standard in high-end FPGAs, such as, the Altera Stratix II/III/IV [Altera 2008a; 2008b; 2008c] and Xilinx Virtex-5 [Xilinx 2008a; 2008b]. DSP blocks contain no programmable routing, and thus incur minimal overhead; moreover, their operations are implemented using ASIC logic gates rather than LUTs. These two factors give them a distinct advantage over FPGA general logic for parallel multiplication and MAC.

Altera's DSP blocks perform addition/subtraction, multiplication/MAC, shifting, rounding, and saturation for a variety of wordlengths (9-, 12-, 18-, and 36-bit) [Altera 2008b; 2008c; 2008d]. The Xilinx DSP48 offers similar functionality, but with different bitwidths, for example, 25×18-bit multiplication, and also includes several bitwise logical operations. The DSP48E variant allows for SIMD addition, subtraction, and bitwise logical operations, but cannot use the multiplier at the same time [Xilinx 2008a]. An FPCT, in contrast, would not replace partial product generation, rounding, saturation, or shifting [Altera 2008b; 2008c]; these operations would use the general FPGA logic.

Kuon and Rose [2007] observed that 5×5-bit multiplication was faster on the general logic of the FPGA than on the larger, fixed-bitwidth parallel multiplier in a DSP block. Unlike DSP blocks, an FPCT is programmed to match the bitwidth of the input operands; also, several small-bitwidth operations can be mapped onto one FPCT. More generally, Beuchat and Tisserand [2002] described a method to build hybrid multipliers using a combination of LUTs and DSP blocks to gracefully handle bitwidth mismatches.

3.2 Carry Chains

FPGA logic cells are organized in clusters connected with local routing, which is faster than routing between cells in disparate clusters. Moreover, carry chains have been introduced to improve performance for common arithmetic operations, namely carry-propagate addition [Cherepacha and Lewis 1996; Kaviani et al. 1998; Hauck et al. 2000; Leijten-Nowak and van Meerbergen 2003; Frederick and Somani 2006] and carry-save addition [Parandeh-Afshar et al. 2008a]. In addition to programmable LUTs, each logic cell is augmented with some dedicated arithmetic logic (e.g., part of a CPA), and direct connections to neighboring logic cells in the cluster. The fixed function logic reduces the delay of the operations significantly compared to using LUTs, while the direct connections wholly eliminate routing delays. Logic cells augmented with carry chains have a performance advantage for the operations implemented by the carry chains.

The Altera Stratix II/III/IV [Altera 2008a; 2008b; 2008c] and Xilinx Virtex-5 [Xilinx 2008a; 2008b] logic blocks contain carry chains that perform carry-propagate addition. The carry chain in Altera's Adaptive Logic Module (ALM) is a ripple-carry adder, with a carry-select feature: In arithmetic mode, the ALM can be configured as an adder, counter, accumulator, or comparator. In shared arithmetic mode, the ALM is configured as a 3-input (ternary) adder: The LUTs are configured as a carry-save adder, whose outputs are fed into the appropriate inputs of the ripple-carry adder. Likewise, the Virtex 5 carry chains can also perform ternary addition [Cosoroaba and Rivoallon 2006].

3.3 Compressor Tree Synthesis on FPGAs

Poldre and Tammemae [1999] synthesized compressor trees using the carry chains on Xilinx Virtex FPGAs. Their method was tied to the Virtex logic cell architecture and carry chain, which has evolved considerably since then.

Support for ternary CPAs in FPGA logic cells [Altera 2006] was motivated by the assumption that carry chains make adder trees faster than compressor trees on FPGAs. Parandeh-Afshar et al. [2008b; 2008c], however, observed that GPCs with six inputs and three or four outputs map have shorter critical path delays compared to ternary adder trees; however, this approach increased the number of logic cells required.

Parandeh-Afshar et al. [2008a] integrated a second carry chain into an ALM, permitting its configuration as a 6:2 compressor, a circuit similar to a 6:3 counter, but employing a short, bounded-length carry chain. 6:2 compressors achieve area utilization comparable to ternary adders with higher compression ratios than 6-input GPCs. The use of 6:2 compressors in conjunction with GPC mapping achieves a tighter placement, which reduces wirelength, and, in turn, routing delays.

3.4 Predecessors to the FPCT

A *Field Programmable Counter Array (FPCA)* [Brisk et al. 2007] is a predecessor of the FPCT presented in this article. An FPCA has a similar architecture to an FPGA, but the logic cells are hierarchically designed *m*:*n* counters, rather than LUTs, which are connected through a global routing network. An FPCA is intended for integration within a larger FPGA. Compressor trees are synthesized on the FPCA, while the remainder of the circuit is synthesized on the FPGA.

This article is an extension of prior work by Cevrero et al. [2008], who introduced the FPCT (then-called "FPCA-2.0"). Counters in the FPCT are organized as a tree with a limited amount of programmable interconnect compared to the array organization. When a compressor tree is synthesized on the general logic of an FPGA or on an FPCA, routing delays are inevitable. The FPCT, in contrast, eliminates much of the routing overhead within a compressor tree while providing the correct building blocks, *m*:*n* counters.

4. FPCT ARCHITECTURE

4.1 Motivating Example

The FPCT architecture was motivated by a simple pattern observed for multi-input addition, shown in Figure 4(a). Let us add *k* *n*-bit integers using a

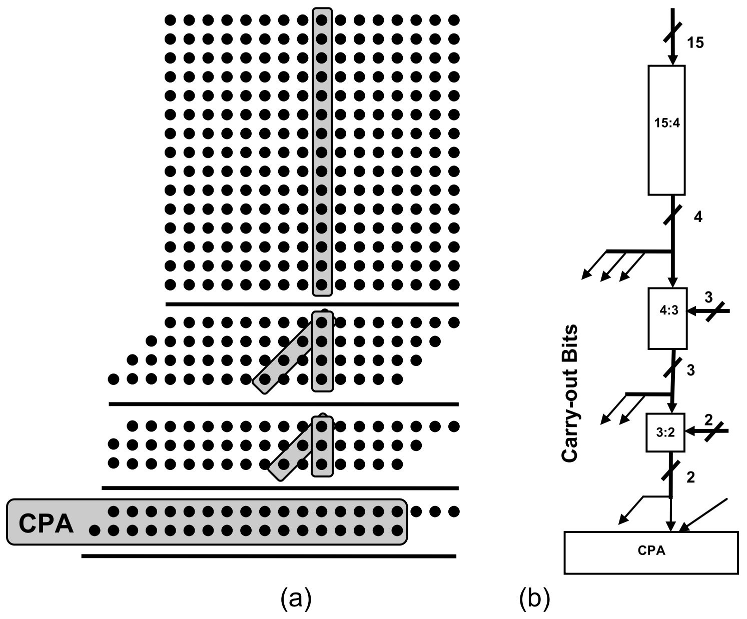


Fig. 4. (a) A set of columns of fixed size can be compressed with a collection of counters in descending size that follows a specific pattern: Each m:n parallel counter is followed by an n-input parallel counter, stopping when $n = 2$; and a CPA is used to perform the final addition; (b) a CChain.

compressor tree. Assume that k=15 and that $n >> k$. Initially, a 15:4 counter compresses each 15-bit column.

Consider the i^{th} column of bits; the 15:4 counter propagates four output bits of rank i, $i+1$, $i+2$, and $i+3$ to the next level of the tree. In the i^{th} column of the following level there are now four bits, produced by the 15:4 counters covering the $(i-3)^{rd}$,$(i-2)^{nd}$, $(i-1)^{st}$, and i^{th} columns of the first level. All remaining columns can be covered using 4:3 counters; at the following level, there are three bits per column, so each column is covered with a 3:2 counter; finally, a CPA adds the remaining bits.

Figure 4(b) shows a vertical chain of descending counters called a *Counter Chain (CChain)* based on the pattern in Figure 4(a). The CChain is an arithmetic component in the same general class as counters, GPCs, and compressors. This particular CChain is a 15:2 compressor that takes six additional carry-in bits (all of rank 0) and produces six carry-out bits: three of rank 1, two of rank 2, and one of rank 3. The 15:4 counter propagates carry bits to the 4:3 counters of rank i, $i+1$, $i+2$, and $i+3$; likewise, the 4:3 counter propagates carry bits to the 3:2 counters of rank i, $i+1$, and $i+2$; and lastly, the 3:2 counter respectively propagates its sum and carry bits of rank-i and rank-$(i+1)$.

Figure 5 shows a cascaded array of CChains that form a small compressor tree. Figure 5 illustrates the pattern of carry bits that are propagated from one CChain to those that follow it, and to the CPA. The four CChains shown in Figure 5 compute the least significant four bits of the sum of fifteen four-bit

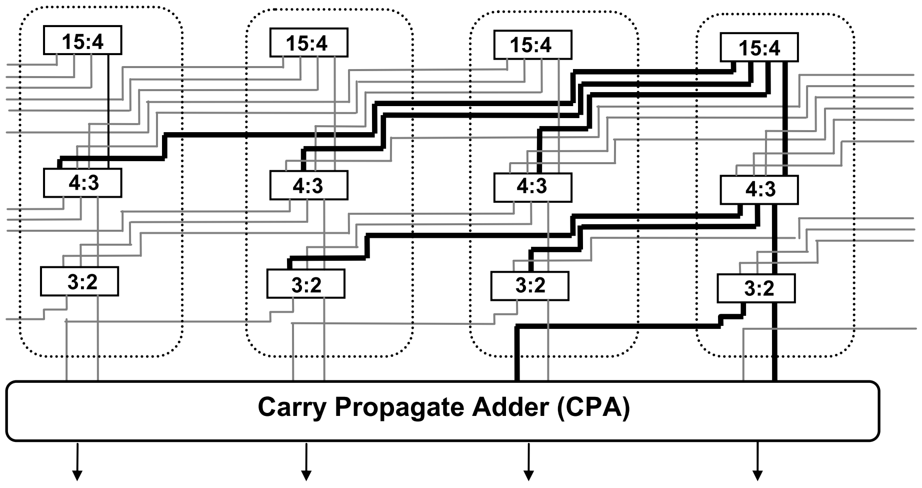


Fig. 5. Cascading CChains to propagate carry bits.

Table I. For a Fixed Column Size m in Figure 3, the Number of Levels in the Compressor Tree and the Descending Sequence-Specific Counters Required

m	Number of Levels	CChain
$m = 1$	0	
$m = 2$	1	CPA
$m = 3$	2	(3:2), CPA
$4 \leq m \leq 7$	3	(m:3), (3:2), CPA
$8 \leq m \leq 15$	4	(m:4), (4:3), (3:2), CPA
$16 \leq m \leq 31$	4	(m:5), (5:3), (3:2), CPA
$32 \leq m \leq 63$	4	(m:6), (6:3), (3:2), CPA
$64 \leq m \leq 127$	4	(m:7), (7:3), (3:2), CPA
$128 \leq m \leq 255$	5	(m:8), (8:4), (4:3), (3:2), CPA

integers. To compute the remaining bits, five additional CChains are needed to add the carry-out bits produced by the three leftmost CChains in Figure 5; all inputs to the 15:4 counters in the subsequent CChains are set to 0.

Given a positive integer m, the generation of a CChain starting with an $m : n$ counter is straightforward and deterministic. For a given value of m (15 in the case of Figures 4 and 5), we begin with an $m : n$ counter; the second counter in the chain has n inputs. Applying the same reasoning recursively, the number of output bits of the j^{th} counter in the CChain will be the number of input bits of the $(j + 1)^{st}$ counter. This process repeats until two bits remain, which are connected to the CPA.

Table I lists the CChains generated starting with $1 \leq m \leq 255$.

4.2 FPCT and CSlice Architecture

The FPCT is a one-dimensional array of building blocks called *Compressor Slices (CSlices)*. A CSlice is a CChain that has been augmented with a variety of programmable features that increase its flexibility. Figure 6(a) and 6(b),

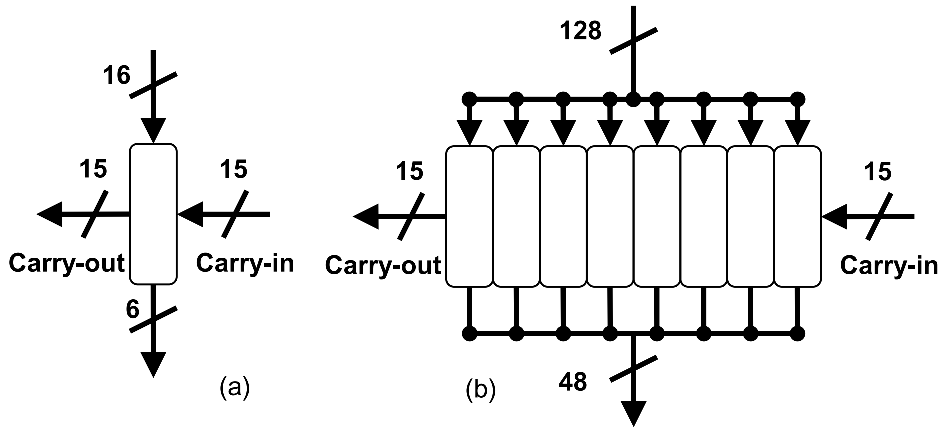


Fig. 6. High-level block diagram of: (a) a CSlice; and (b) an 8-CSlice FPCT.

respectively, show high-level block diagrams of the CSlice and FPCT. Each CSlice has 31 input bits, 15 of which are carry-in bits from preceding CSlices, and 21 output bits, 15 of which are carry-out bits. Each FPCT has 143 input bits, 128 of which are directly connected to the CSlice inputs, and 15 of which are the carry-in bits of the first CSlice; similarly, each FPCT has 63 output bits, 48 of which are outputs of the eight CSlices, and 15 of which are the carry-out bits of the last CSlice. The carry-in and carry-out bits permit the concatenation of several FPCTs together; this is discussed in greater detail in Section 4.3.

Figure 7 illustrates the architecture of a simplified CSlice that produces 4, rather than 6, output bits. In Figure 7, the lower part of the CChain is replicated once and the CPA produces two output bits; in actuality, the lower part is replicated twice and the CPA produces three output bits. A detailed description of the CSlice components follows.

A CSlice has 16 input bits (ignoring carry-in bits). An *Input Configuration Circuit (ICC)*, shown in Figure 8(a), allows the user to independently drive each input bit to 0. This feature is useful when there are fewer than 16 bits to sum. The alternative is to drive the CSlice inputs to 0 using the FPGA's routing network. We have opted for the ICC in order to save routing resources. The delay overhead (one AND gate per input wire) and area overheads (16 AND gates and 16 configuration flip-flops) are insignificant.

The primary CChain begins with a 31:5 counter. The *GPC Configuration Circuit (GPCCC)* allows the user to program the 31:5 counter as a variety of GPCs with up to 16 input bits and up to 5 output bits. The GPCCC expands the 16 input bits to 31, which are then consumed by the 31:5 counter, permitting the counter to sum bits of varying rank. Many GPCCCs are possible. Figure 8(b) shows a GPCCC that allows each input bit to be configured as rank-0 or rank-1 using one configuration bit. Figure 8(c) shows a GPCCC that allows each input bit to be configured as rank-0, 1, or 2 using two configuration bits.

Suppose we have six bits of rank i, and ten of rank i+1 to add. One possibility is to use two consecutive CChains: one for the rank i bits, and the other for the

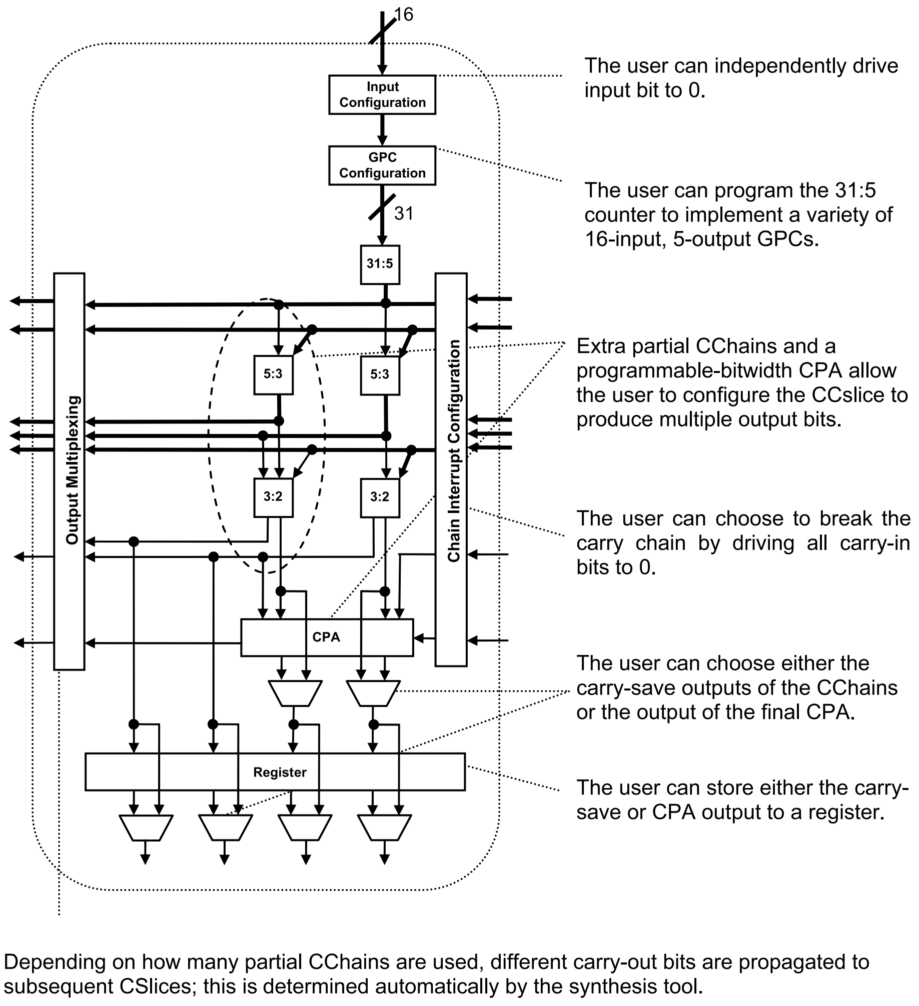

16
Input Configuration
GPC Configuration
31
31:5
5:3
5:3
3:2
3:2
CPA
Register
Output Multiplexing
Chain Interrupt Configuration
The user can independently drive input bit to 0.
The user can program the 31:5 counter to implement a variety of 16-input, 5-output GPCs.
Extra partial CChains and a programmable-bitwidth CPA allow the user to configure the CCslice to produce multiple output bits.
The user can choose to break the carry chain by driving all carry-in bits to 0.
The user can choose either the carry-save outputs of the CChains or the output of the final CPA.
The user can store either the carry-save or CPA output to a register.
Depending on how many partial CChains are used, different carry-out bits are propagated to subsequent CSlices; this is determined automatically by the synthesis tool.

Fig. 7. CSlice architecture.

rank $i + 1$ bits. Alternatively, we can replace each rank $i + 1$ bit with two rank i bits, totaling 26 rank i bits, which can be summed with one CChain. We could use the FPGA routing network to connect each rank $i+1$ bit twice to the CSlice input; instead, the GPCCC performs the aforementioned expansions, reducing pressure for resources in the routing network.

After the 31:5 counter, the CSlice contains replicated partial CChains, starting with 5:3 counters. Replicating the CChains allows the user to configure the CSlice to produce multiple output bits, and then program the CPA accordingly. Although Figure 7 shows a CSlice with two partial CChains, our implementation actually employs three.

The 31:5 counter dominates the CSlice area; that of the replicated partial CChains is insignificant. Producing multiple output bits per CSlice, when

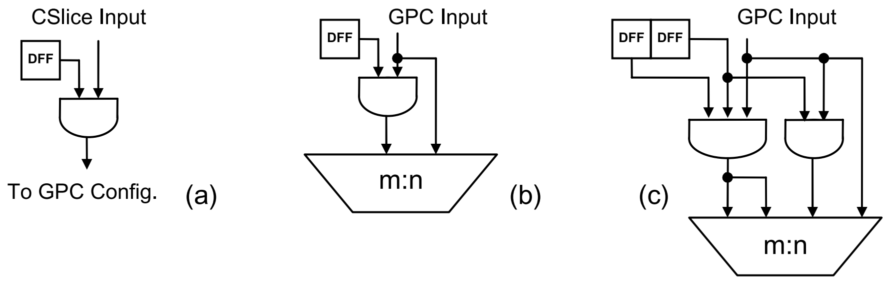


Fig. 8. (a) ICC for one input bit; (b) an example of a GPCCC that allows each input bit to the counter to be configured as rank-0 or rank-1; (c) an example of a GPCCC that allows each input bit to the counter to be configured as rank-0, rank-1, or rank-2.

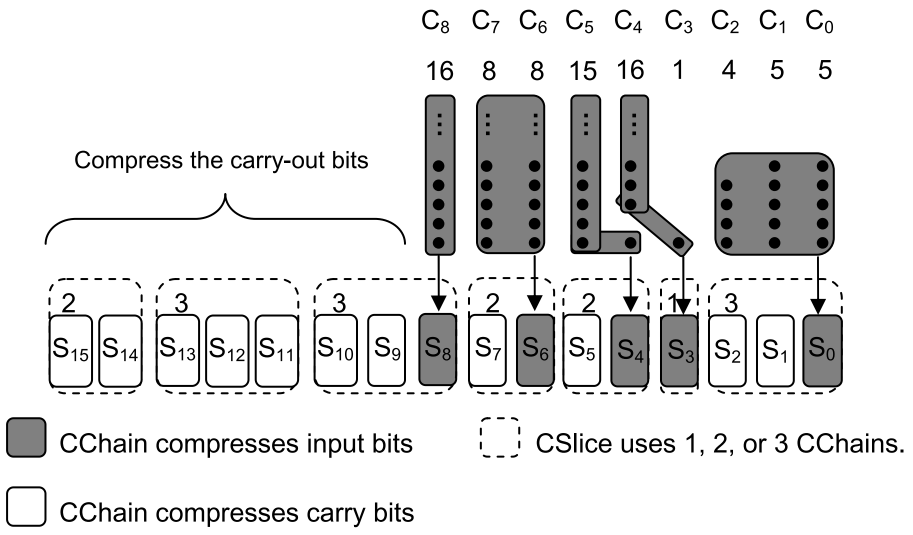

C8 C7 C6 C5 C4 C3 C2 C1 C0
16 8 8 15 16 1 4 5 5
Compress the carry-out bits
2 3 3 2 2 1 3
S15 S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0
CChain compresses input bits
CSlice uses 1, 2, or 3 CChains.
CChain compresses carry bits

Fig. 9. Increasing the number of output bits produced by a CSlice improves its utilization.

possible, reduces the number of CSlices required to synthesize a compressor tree on an FPCT. This improves significantly the area utilization of the FPCT. Figure 9 shows an example, in which nine columns of dots $C_0, \ldots, C_8$ are compressed using sixteen CChains, $S_0, \ldots, S_{15}$. The 31:5 counter is only used in five of the sixteen CChains. If each CSlice contains one CChain and produces one output bit, then sixteen are needed in Figure 9; alternatively, if each CSlice contains three CChains, two of which are partial, then seven CSlices are needed; this significantly improves resource utilization.

A different set of carry-out bits will be propagated to subsequent CSlices, depending on the configuration of the CSlice; an output multiplexing layer, shown on the lefthand-side of Figure 7, is configured by the synthesis tool to select the correct carry-out bits to propagate, depending on how many output bits are produced by the current CSlice.

The CSlice can produce several different types of outputs: the sum of the input bits using the compressor tree and CPA together, or just the carry and save outputs of the compressor tree, bypassing the CPA. Each CSlice can optionally store either the CPA or carry-save outputs to a register. Similar to an FPGA logic cell, a multiplexor following the register can select either the combinational or sequential output.

Lastly, the *Chain Interrupt Configuration Circuit (CICC)* can be programmed to drive all carry-in bits of the CSlice to 0. This is useful for FPCTs with many CSlices where several independent compressor trees are mapped onto one FPCT; the CICC prevents the carry-out bits from one compressor tree from propagating into the next. For example, an $n \times n$-bit multiplier produces a $2n$-bit output. If only n output bits are needed, then the CICC suppresses the propagation of carry bits to additional CSlices.

4.3 Configurable CPA

A CPA adds the sum and carry output bits of a compressor tree. For parallel multipliers in ASIC technologies, the arrival time of the input bits at the final CPA is nonuniform [Oklobdzija and Villeger 1995]. Stelling and Oklobdzija [1996] presented a heuristic to tailor the final CPA to the arrival time of the bits. Although arrival times may also be nonuniform in our case, we cannot know them precisely a priori, as many different multi-input adders and multipliers will be synthesized on an FPCT.

Carry-select adders can exploit nonuniform arrival times to reduce the critical path delay. In an n-bit carry select adder, the bits are partitioned into k groups. Within each group, the bits to sum are added twice in parallel: The first assumes a carry-in bit of 0 and the second assumes a carry-in bit of 1. Starting with the least significant group, the carry-out bit from the preceding group selects the correct output from the current group, along with the carry-out to the following group. Adding the k groups in parallel is significantly faster than n-bit ripple-carry addition; instead, the critical path includes the delay of (n/k)-bit ripple-carry addition, plus the resolution of the carry-out bits for each group.

The CPA for each CSlice is a configurable carry-select adder. Firstly, the group size n/k is configurable; secondly, it can be configured to produce 1, 2, or 3 output bits. As the group size is not known a priori, the CPA must implement different functionality depending on whether or not each CSlice begins a new group.

Figure 10 shows the CPA. The register *rank* contains the number of pairs of bits to sum; depending on its value, different carry-out bits from the ripple-carry adder are propagated to the subsequent CPA. A flip-flop *(DFF)* is set to 0 if the CPA is the first in a group, and 1 otherwise. If the CPA is the first in a group, then the carry-in bit to each ripple-carry adder is 0 or 1, and the bit that selects the correct output is the carry-out bit of the preceding group.

Otherwise, the carry-in bit to each ripple-carry adder is the carry-out bit of the ripple-carry adder in the preceding CSlice, and the selection bit is propagated from the preceding CPA in the same group.

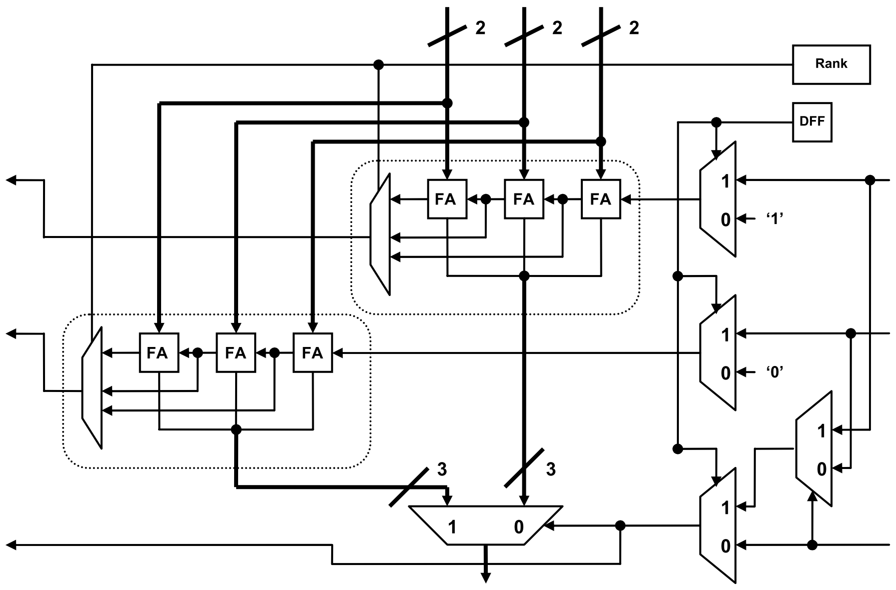


Fig. 10. Programmable carry-select CPA architecture for a CSlice that produces three output bits.

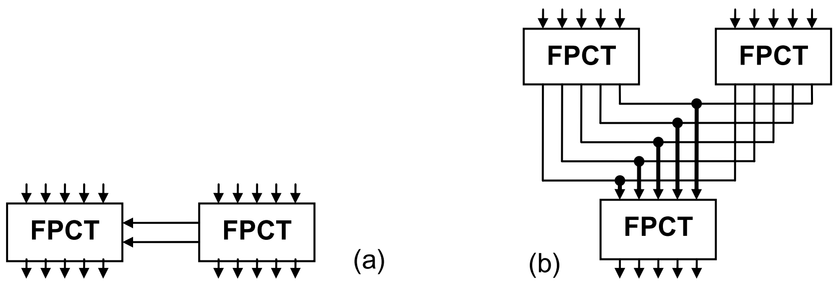


Fig. 11. Multi-FPCT configurations: (a) a horizontal configuration; (b) a vertical configuration.

4.4 Multi-FPCT Configurations

Several DSP blocks can be wired together to perform large-bitwidth multiplication and MAC operations; likewise, FPCTs may be combined in a similar fashion to implement large compressor trees.

A *horizontal configuration*, shown in Figure 11(a), occurs when the number of CSlices required for a compressor tree exceeds the number of CSlices available in an FPCT. In this case, the carry-out bits of the last CSlice of one FPCT connect to the carry-in bits of the first CSlice in the next through the FPGA programmable routing network.

A *vertical configuration*, shown in Figure 11(b), occurs when the number of bits, per column, exceeds the number of inputs to a CSlice. If the input banwidth of a CSlice is *m* and each column has *km* bits, then *k* CSlices will be required to compress this column. If each FPCT produces the output in carry-save form, then there will be $2k$ output bits after this compression step; a second layer of FPCTs adds the remaining bits.

Vertical configurations can be pipelined by using the registers shown in Figure 7; horizontal configurations require an additional (bypassable) register to store the carry outputs of the final CSlice. If multiple pipeline stages are required between two FPCTs, additional pipeline registers can be synthesized on the general logic of the FPGA.

5. FPCT SYNTHESIS

5.1 FPCT Mapping: Problem Formulation

Let $B = \{b_0, \ldots, b_{M-1}\}$ be a set of bits to sum, where $rank(b)$ is the rank of bit $b \in B$. The bits are organized into k columns: $C = \{C_0, \ldots, C_{k-1}\}$; $C_i = \{b \in B | rank(b) = i\}$ is the set of bits of rank i. The bits in B are ordered so that for each pair of bits, b_j and b_{j+1}, $rank(b_j) \leq rank(b_{j+1})$. Thus, the first $|C_0|$ bits belong to C_0, the next $|C_1|$ bits belong to C_1, etc. Let the target FPCT be comprised of k CSlices: $S = \{S_0, \ldots, S_{k-1}\}$. The problem remains the same if there are fewer columns than CSlices. Let R_{in} be the rank of the input (Section 2.3), which is set to 3 in our experiments, and N be the number of CSlice inputs.

The output of the mapping procedure is a function $f : B \rightarrow \{\perp, 0, 1, .., k-1\}$ that describes the mapping of bits onto CSlices. For bit $b \in B$, $f(b) = \perp$ if b is not mapped to a CSlice; otherwise, $f(b) = i$, $0 \leq i \leq k-1$. Let $B_i = \{b \in B | f(b) = i\}$ is the set of bits mapped onto CSlice S_i; $B_\perp = \{b \in B | f(b) = \perp\}$ is the set of unmapped bits.

For each bit b_j we define the quantity Δ_j as follows.

$$\Delta_j = \begin{cases} 0 & if \;\; b_j \in B_\perp \\ rank\left(b_j\right) - f\left(b_j\right) & if \;\; b_j \in B - B_\perp \end{cases} \tag{2}$$

A legal mapping solution satisfies the following two constraints.

$$|B_i| \leq N \qquad 0 \leq i \leq k-1, \text{ and} \tag{3}$$

$$0 \leq \Delta_j \leq R_{in} - 1 \qquad 0 \leq j \leq M-1 \tag{4}$$

Constraint (3) ensures that the number of bits assigned to a CSlice does not exceed N, the number of CSlice input ports. Constraint (4) ensures that the rank of a bit must not be smaller than the rank of a CSlice. Clearly, we can map a bit of larger rank to a CSlice of smaller rank by connecting the bit to multiple counter inputs (as permitted by the GPCCC); however, we cannot map a bit of smaller rank to a CSlice of larger rank by connecting it to less than one input. For example, a CSlice S_2 (of rank 2) counts values 0, 4, 8, 12, ...; the CSlice does not have sufficient granularity to count all the values of a rank-1 bit: 0, 2, 4, 6, 8, ..., or a rank-0 bit: 0, 1, 2, 3, ... Constraint (4) also ensures that the difference between the rank of each bit b_j mapped onto CSlice S_i does not exceed $R_{in} - 1$. For example, if $R_{in} = 3$, then S_i can only take bits from three columns: C_i, C_{i+1}, C_{i+2}, or equivalently, bits whose ranks are i, $i+1$, or $i+2$.

The optimal solution to the FPCT mapping problem minimizes the height of a compressor tree built from FPCTs with vertical connections. The complexity of this problem has not yet been analyzed; it may or may not be NP-complete.

5.2 FPCT Mapping Heuristic

We are given a set of columns of bits to be added, the number of CSlice inputs, N, and the rank of the input, R_{in}, defined in the previous section. We use a modified version of the GPC mapping heuristic [Parandeh-Afshar et al. 2008b] to cover all of the input columns with GPCs, and to map the covered bits onto CSlices in one (or more) FPCTs.

The GPC mapping heuristic requires a library of GPCs. We systematically enumerate all GPCs having at $N = 16$ input bits, at most 5 output bits, and whose input bits have at most $R_{in} = 3$ different ranks. Limiting the number of output bits to 5 satisfies the constraint that a 31:5 counter can be used, as the sum of the input bits, weighted by rank, does not exceed $31 = 2^5 - 1$. The library, after enumeration, contained 58 GPCs.

Let $G = \{G_1, \ldots, G_P\}$ be the covering, that is, each GPC G_i covers at most 16 bits spanning three columns. Limiting the number of bits per GPC satisfies constraint (3), since each GPC will be mapped onto one CSlice. The limit of 3 columns per GPC ensures that all GPCs satisfy constraint (4). The *rank* of a GPC, $R(G_i) = min\{rank(b)|b\in G_i\}$, is the minimum rank among all bits in G_i. Let K be the number of CSlices in the FPCT ($K = 8$ in our experiments). We must pack the GPCs found by the covering into groups of at most K GPCs, such that each group can be mapped onto an FPCT. The packing process must satisfy the following constraints: (i) If $R(G_i) = R(G_j)$ then G_i and G_j cannot be part of the same group; (ii) If $R(G_j) > R(G_i)$, G_i and G_j can be packed into consecutive CSlices in the same FPCT only if $1 \le R(G_j) - R(G_i) \le 3$.

To pack the different GPCs in the covering onto FPCTs, we find chains of GPCs that satisfy constraint (ii) given before. If we find a chain that contains more than K GPCs, then a horizontal configuration is used. After each chain is identified, the GPCs in the chain are removed; then the process repeats and a new chain among the remaining GPCs is found.

The process stops after all GPCs have been mapped to an FPCT CSlice. In the example of Figure 9, there is a single chain of five GPCs, which are mapped directly onto the CSlices (shown with dashed lines); rank-1, -2, and -3 configurations are all used.

Two (or more) short chains can be mapped onto an FPCT by breaking the carry propagation with the CICC. After the initial mapping phase, we look for pairs of short chains that can utilize unused CSlices on FPCTs that have already been allocated. Figure 12 shows an example. The number of output bits produced by each CSlice is determined based on which GPC is mapped onto it. If the GPC is an $m : n$ counter, then an ORC of 1 is appropriate; otherwise, rank-2 or rank-3 configurations are selected.

If there is at most 1 bit per column, we are done; otherwise, a vertical configuration is required. In this case, we configure each CSlice in the previous level to produce the sum and carry outputs to avoid the use of CPAs internally

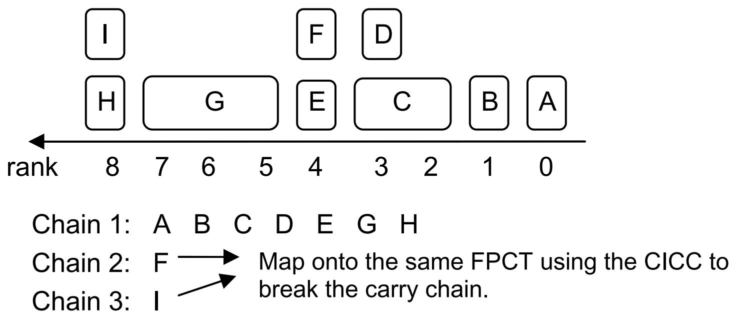


Fig. 12. Example of packing GPCs into CSlices. Three chains are found, two of which can be mapped onto the same FPCT.

within a larger compressor tree. Then we repeat the mapping process for the resulting columns of bits.

5.3 Pipelining via ASAP and ALAP Scheduling

In pipelining mode, the output of the FPCT is always stored in its *internal* pipeline register. All other pipeline stages must be stored in the logic blocks of the FPGA general logic; these are called *external* pipeline registers. A direct connection from one FPCT to the next connects an internal pipeline registers directly to the input of another FPCT.

Figure 13 shows two schedules of a multi-FPCT configuration with horizontal and vertical connections. Figure 13 also shows the bitwidth of the value stored in each pipeline register. The schedule in Figure 13(a) is computed using the *As Late As Possible (ALAP)* heuristic and requires 640 external flip-flops. The schedule in Figure 13(b) is computed using the *As Soon As Possible (ASAP)* heuristic and requires 480 external flip-flops. The latencies of both schedules are the same.

THEOREM 1. *ASAP scheduling minimizes the size of the external pipeline registers.*

PROOF. Assume to the contrary that a non-ASAP schedule minimizes the number of external flip-flops. Then there must be some FPCT F that could be scheduled one time-step earlier than its current time. It follows that the input to F is stored in an external pipeline register; it would not be possible to move F to the preceding step if a predecessor P of F was scheduled in the preceding cycle. Moving F to the preceding cycle advances the external pipeline register ahead of F. As this was the only transformation, it follows that no successor S of F is scheduled one cycle following F. This guarantees that the output of F's internal register is written to the pipeline register that was advanced in front of F. Since F compresses the number of bits, the bitwidth of the external pipeline register is reduced by advancing it in front if F, contradicting the assumption that the non-ASAP schedule minimized the number of flip-flops required for external pipeline registers. □

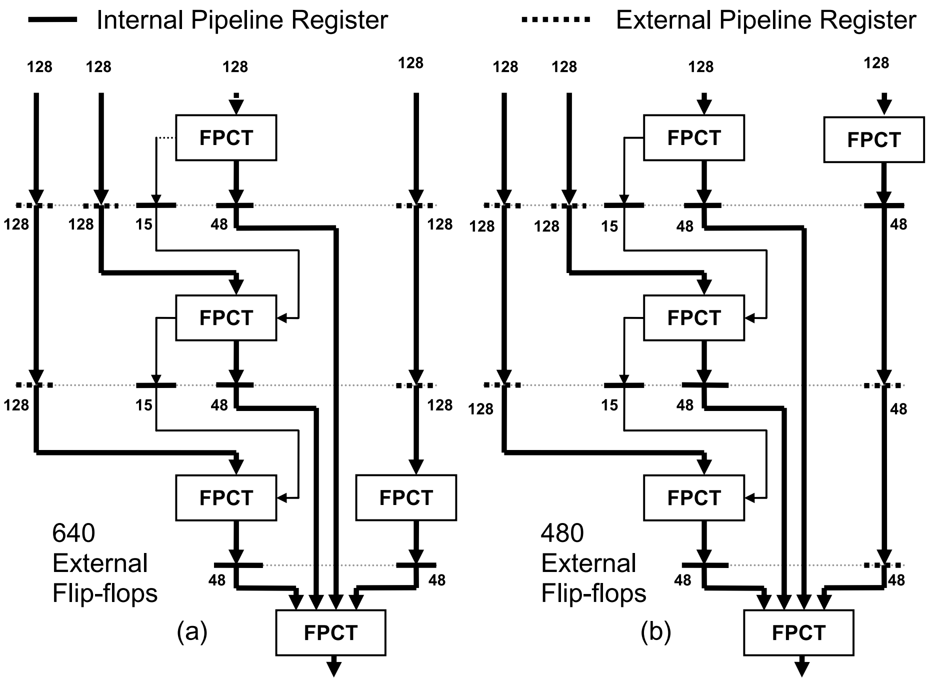


Fig. 13. Pipelined multi-FPCT configuration scheduling using: (a) ALAP; and (b) ASAP scheduling.

External pipeline registers that precede the FPCT in the ALAP scheme can be absorbed into the combinational logic that precedes the compressor tree via retiming. This eliminates the cost of the register, as FPGA logic blocks contain bypassable flip-flops; so, registering the output of a logic layer comes at no additional cost in terms of hardware. On the other hand, this is not possible when using the ASAP scheme, as there is no combinational logic between FPCTs, for example, in Figure 13(b). Consequently, the best strategy for pipeline register insertion is to try ASAP scheduling and ALAP scheduling coupled with retiming; the result that minimizes the number of logic cells is then chosen. Our experiments, however, compare just the straightforward ASAP and ALAP schemes.

6. EXPERIMENTAL SETUP

6.1 FPCT Synthesis

The CSlice architecture and FPCT were modeled in VHDL; each FPCT was built from eight abutted CSlices. Correctness was verified using Mentor Graphics Modelsim; synthesis was performed with Synopsys Design Compiler and Design Vision; placement and routing was performed by Cadence Design Systems' Silicon Encounter tool. A 90nm Artisan standard cell library based on a TSMC design kit was used. Floorplanning was performed with the goal of minimizing the routing delay.

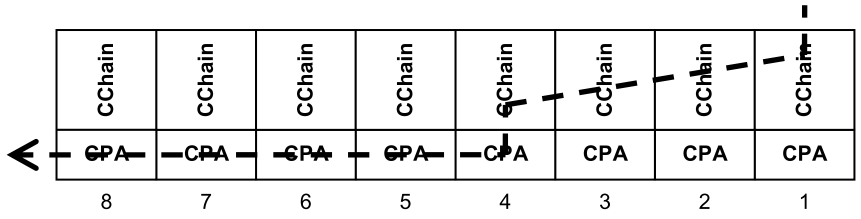


Fig. 14. Delay profile of an 8-CSlice FPCT.

To achieve the best synthesis results, we decoupled the compressor tree portion of the CSlice from the CPA and optimized these two structures separately. If the CSlice is synthesized as a standalone block, the CPA carry-in to carry-out delay is negligible compared to the delay through the compressor tree; however, this delay becomes significant when several CSlices are concatenated. Decoupling the two parts of the CSlice permits separate optimization of the compressor tree and the CPA. In the case of the CPA, the 3-bit ripple-carry adder shown in Figure 10 was replaced with a larger but more efficient adder structure, namely, the fastest possible 1-, 2-, and 3-bit adders that the optimizer could find. The result was 60% larger than the ripple-carry adder, but $5\times$ faster.

Figure 14 shows the delay profile for an 8-CSlice FPCT after placement and routing. The critical delay of the compressor tree spans four CSlices, while the remainder of the delay is through the CPA. Let D_T be the delay of the compressor tree, D_{CPA} be the delay of the CPA in a single CSlice, and N be the number of CSlices. Then, for $N \geq 4$, the critical path delay is given by: $D_{Critical} = D_T + (N - 4)D_{CPA}$. $D_{Critical}$ is an upper bound on the contribution of one FPCT to the critical path delay. For example, the FPCT can perform a low-bitwidth operation that is mapped onto fewer than eight CSlices. The delay depends on the configuration on the FPCT. For an 8-CSlice FPCT, the static timing analyzer reported $D_{Critical} = 1.595$ ns, that is, a maximum frequency of 627 MHz. The compressor tree delay in the first four CSlices accounts for 79% of $D_{Critical}$, while the CPA delay through the last four accounts for the other 21%. The area of each CSlice was 3,500 μm^2; the area of an 8-CSlice FPCT was 28,000 μm^2.

6.2 Comparison with DSP Blocks and LABs

We selected the Altera Stratix II FPGA to evaluate the FPCT. Each Stratix II DSP block is divided into half-DSP blocks [Altera 2008a].We implemented a block to mimic the functionality of the half-DSP block, synthesized, and placed-and-routed it using the same design flow. Both Xilinx and Altera publish maximum clock frequencies of 550 MHz for their DSP blocks [Altera 2008b; Xilinx 2008b].

In our experimental infrastructures, we assume one FPCT block can replace one half-DSP block in a Stratix-series chip. To validate this assumption: (1) the FPCT should have comparable I/O requirements to the half-DSP block; and (2) the FPCT's area should not exceed the area of a half-DSP block.

Table II. Comparison of the Altera Stratix II Half-DSP Block and an 8-CSlice FPCT

	Half-DSP Block	8-CSlice FPCT
Frequency	550 MHz	$\geq$ 627 MHz
Area	$> 28{,}000\ \mu\text{m}^2$	$28{,}000\ \mu\text{m}^2$
Input Pins	144	143
Output Pins	72	63

Table II quickly summarizes the result of this comparison. The half-DSP block runs at 550 MHz, while the FPCT has a variable frequency of at least 627 MHz, depending on its configuration. This guarantees no performance advantage, however, as the critical path may be influenced by general logic in conjunction with the DSP block/FPCT, or may simply be in another part of the circuit altogether. The I/O requirements of the two blocks are comparable, and we have determined that the half-DSP block is larger than an 8-CSlice FPCT; however, our methodology for comparing their areas warrants further discussion.

We implemented a block to mimic the functionality of Altera's half-DSP blocks. First, we designed the core datapath elements, such as 9×9-bit multipliers and a small adder tree to combine the outputs when 18×18-bit and 36×36-bit multipliers are built using 9×9-bit multipliers as building blocks. These elements were synthesized using the same design flow as the FPCT, which was summarized in the preceding section.

We constrained the synthesizer to target a clock frequency of 550 MHz, knowing that this speed was not likely to be achieved after placement and routing. After placement and routing, the area of the half-DSP exceeded the 8-CSlice FPCT area. In actuality, two additional factors would cause the half-DSP block to be even larger. Firstly, we only synthesized the computational components (multipliers and an adder tree), ignoring, for example, configuration bits, bypassable registers, multiplexers, support for multiply-accumulate operations, and support for multiplication of complex numbers [Altera 2008a]; additionally, the Stratix III and IV DSP blocks support additional operations, such as rounding, saturation, and shifting [Altera 2008b; 2008c]. Secondly, to achieve an actual frequency of 550 MHz after placement and routing, it would be necessary to set the synthesizer with a target frequency higher than 550 MHz, so that the frequency, after the loss due to placement and routing, would be around 550 MHz; doing this, however, would likely further increase the area of the half-DSP block. Since our goal is simply to ensure that the area of the FPCT does not exceed the area of a half-DSP block, it was unnecessary to determine the precise area of the half-DSP block as described earlier.

Figure 15 shows a layout for a composite FPGA, based on the Stratix II architecture. The layout includes both FPCTs and half-DSP blocks, omitting routing resources.

We also designed, synthesized, and placed and routed a standard cell clone of Altera's *Adaptive Logic Module (ALM)*; it is approximately the same size as a CSlice. In Stratix-series FPGAs, ALMs are clustered into *Logic Array Blocks (LABs)*. Each LAB contains ten ALMs and a sparsely populated intra-LAB crossbar for local routing. The LAB area is equal to the area of the ten ALMs plus the area of the crossbar. We estimate that a LAB is larger than an

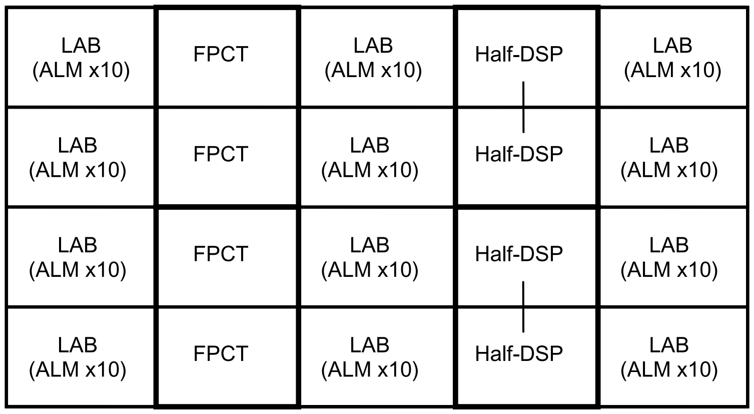


Fig. 15. Illustration of an FPGA, based on the Stratix III, containing FPCTs and half-DSP blocks.

8-CSlice FPCT. To evaluate the FPCT's impact on resource utilization, we will make the conservative assumption that their areas are equal.

6.3 Single FPCT Static Timing Analysis

The critical path of a configured FPCT may be less than the worst-case critical path. The configuration bitstream is derived deterministically after synthesis (Section 5). We wrote as script to emulate a configured FPCT that places constant values of “0” or “1”, at the output of each configuration flip-flop, which closes the majority of input-to-output paths in the FPCT. Next, the FPCT is resynthesized using the “trim unconnected logic” option, to ensure that multiplexors in the FPCT are not optimized away via constant propagation. This yields post-configuration timing information for all paths through the FPCT.

6.4 Multi-FPCT Static Timing Analysis Using Virtual Embedded Blocks

Ho et al. [2006] introduced the *Virtual Embedded Block (VEB)* methodology to evaluate the performance of ASIC cores to be embedded in an FPGA under development. A VEB is instantiated to represent the core under study, and is synthesized and preplaced onto the general logic of the FPGA in the position where the designer intends to instantiate the core. The VEB is designed so that its delay, area, and I/O requirements are comparable to the core under study. Carry chains, whose timing information is precise and well-understood, are the primary function implemented by the VEBs; the length of the carry chain is adjusted to model the delay of the core. Next, a circuit containing an operation to be mapped onto the core is synthesized onto the FPGA, with the VEB's logic function replacing the operation. This gives a precise estimate of the performance of the system using the core in the VEB's preplaced location. As the area and I/O bandwidth of the VEB are comparable to the core's, the placement and routing of the circuit would not change if the core replaced the VEB. Since the VEB and the core have equal delays, the overall delay reported by the system is the same as if the core replaced the VEB.

There is one detail in which our application of the VEB methodology different from Ho et al.'s: The delay of each FPCT output bit varies depending on the configuration. We used a single VEB to evaluate each FPCT configuration. Starting with a detailed timing report, each path through the FPCT was broken into segments at the inputs and outputs of the VEB. First, the delay of the VEB is subtracted from the total delay of each path; second, the delay of the corresponding path through the FPCT is added to the path, effectively compensating for the mismatch in delay between the FPCT and VEB. This is repeated for all VEB instances on all paths. Afterwards, all paths now contain an accurate estimate of the delay through the FPCT.

Ho et al. verified the VEB methodology using an 18×18-bit multiplier in a Xilinx Virtex II FPGA. The error in their delay estimates, compared to the use of an actual multiplier in the same system, ranged from 0–11%. The largest errors reported by Ho et al., +11% and +10% respectively, were for 68×68- and 136×136-bit multiplication, which are larger than the multipliers studied here. Most of the error in measurement occurred in the routing network. In this case, the difference occurs because the VEB cannot be placed in the exact location of the multiplier, as there is no soft logic there. This changed the placement of the circuit near the multiplier and altered the routing delays accordingly. For most benchmarks, Ho et al. reported logic delay errors of around 0.1%. Their three largest logic delay errors were for 34×34-, 68×68-, and 136×136-bit multipliers, 1.9%, 3.4%, and 3.4%, respectively.

Two potential sources of error that Ho et al. did not consider also exist. The first involves positional constraints on the I/O interface of the VEB. For example, in a DSP block, input bit 0 is beside bit 1, which is beside bit 2, etc., for each input integer. To the best of our knowledge, it is impossible to impose positional constraints on the input bits of the VEB. The flexibility given to the VEB may give an overly optimistic view on the routability of the embedded core. A second source of error involves direct connections between embedded cores. Many multipliers and DSP blocks have direct connections to their neighbors so that they can be combined efficiently to perform higher bitwidth multiplication. In principle, an FPGA containing FPCTs may want to instantiate similar direct connections, for example, from the carry-out of one to the carry-in of another, in order to support efficient horizontal configurations. It is not possible to model or evaluate the costs and benefits of these direct connections using the VEB methodology.

Unlike Ho et al., our placement of the VEBs for the FPCTs mimics precisely where we intend to locate them in the floorplan. The placement of the circuit elements near the FPCT is much more likely to be correct than in their experiments. Based on their experiments, the two caveats mentioned already, and the fact that the FPCT evaluation does not suffer the disadvantage in accuracy with respect to VEB placement, we assume an 11% confidence interval (i.e., [-5.5%, +5.5%]) for the delays reported in Section 7.

Our intention is to give a reasonable acknowledgement of the potential error in our measurements vis-à-vis the results reported by Ho et al.; however, whatever error actually exists in our measurements cannot be confirmed without fabricating the device proposed in this article and comparing the results of this study to measurements on the fabricated device. Confirming the error in our estimates is beyond the scope of this work.

6.5 Benchmarks

We used eleven benchmark circuits to demonstrate the capabilities and performance advantages that FPCTs offer FPGAs vis-à-vis DSP blocks. For the purpose of discussion, we divide the applications into four categories:

—*Multipliers*. 9×9-, 10×10-, 18×18-, and 20×20-bit multipliers were used to compare the FPCT against DSP blocks. The 9×9- and 18×18-bit multipliers match precisely the functionality of the DSP blocks, while 10×10- and 20×20-bit multipliers are examples of mismatches. These experiments attempt to quantify the potential benefit that may be sacrificed if an FPCT replaces a DSP block that is used for multiplication.

—*Larger Circuits Containing Multipliers*. Three composite benchmark circuits that contain several multipliers along with other peripheral arithmetic and logic operations were considered: the computational core of *g*721 speech encoding [Lee et al. 1997], polynomial evaluation using Horner's rule *(hpoly)*, polynomial evaluation, and a larger video processing application *(video mixer)*.[1] Initially, the multipliers can be mapped onto DSP blocks; however, Verma et al. [2008] merge the partial product reduction trees with other addition operations, preventing their usage. Multiplication operations do occur in the critical paths of these benchmarks; however, the critical path delays of the circuit exceed the critical path delay of a DSP block by $3 - 4\times$.

—*Circuits Naturally Containing Adder / Compressor Trees*. This class contains three circuits: 3- and 6-tap FIR filters *(fir3, fir6)*, and one processing element of a variable block size motion estimator for H.264 video coding (*H.264 ME*) [Parandeh-Afshar et al. 2009].These circuits are naturally written with compressor trees or all CPAs clustered together to form adder trees. Because these applications contain no multipliers, they cannot use DSP blocks, but can exploit the FPCT.

—*adpcm Kernel*. Lastly, the *adpcm* kernel belongs in a separate class, as it contains three disparate CPAs, but no multipliers, as shown in Figure 1(a); it cannot exploit DSP blocks, but does benefit from Verma et al.'s transformations, which cluster the three CPAs to form a 4-input adder/compressor tree, as shown in Figure 1(b).

6.6 Methodology

Each circuit was evaluating using three or four different synthesis flows.

DSP / Untransformed. The four multipliers *g*721, *hpoly*, *video mixer*, and *adpcm* were evaluated using this flow. The transformations of Verma et al. were not applied, in order to maximize the use of DSP blocks for multiplication for all benchmarks other than *adpcm*. As *adpcm* does not use DSP

[1]The HDL implementation of Video Mixer has been provided by Synopsys Corporation as part of their Behavioral Optimization of Arithmetic (BOA) software package.

blocks but does benefit from the transformations, it is evaluated without applying the transformations as well. All CPAs are synthesized using carry chains.

The remaining three synthesis flows are evaluated on all benchmarks. The transformations of Verma et al. were applied to *g*721, *hpoly*, *video mixer*, and *adpcm*; these transformations did not affect the four multipliers *fir3*, *fir6*, or *H.264 ME*. In the case of *g*721, *hpoly*, and *video mixer*, the transformations prevent the use of DSP blocks because the partial product reduction trees are merged with other addition operators.

Ternary. All multi-input adders and multipliers are synthesized on soft FPGA logic using ternary adder trees [Altera 2006].

GPC. All multi-input adders and multipliers are synthesized on soft FPGA logic as compressor trees using the GPC mapping heuristic of Parandeh-Afshar et al. [2008b].

FPCT. All multi-input adders and the partial product reduction trees of all parallel multipliers are mapped onto the FPCT.

Ternary and GPC are unrealistic choices for multipliers as long as a sufficient supply of DSP blocks is available. These results are only included for completeness.

6.7 Synthesis and Evaluation

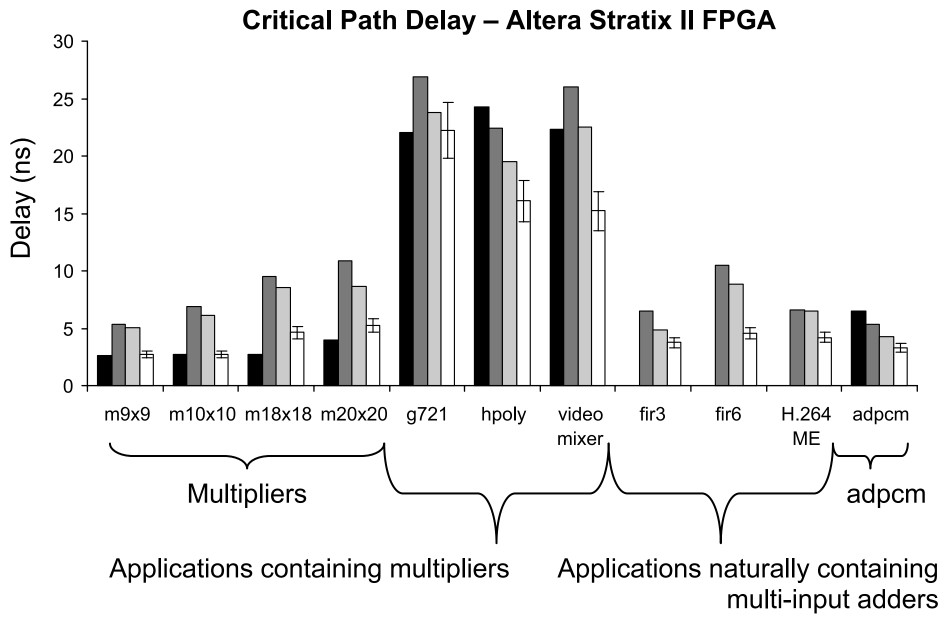
All circuits were synthesized using Altera's Quartus II software targeting a Stratix II FPGA. The delays reported for DSP/Untransformed, Ternary, and GPC in the following section were results reported directly by Quartus II. The methodology outlined in Sections 6.3 and 6.4 was used to evaluate FPCT.

For all benchmarks, other than the four multipliers evaluated on DSP blocks, we used a feature called *virtual pins* to implement the I/O interface. Virtual pins are used when the I/O of a circuit exceeds the I/O capabilities of the FPGA onto which it was mapped. Virtual pins effectively time-multiplex the interface between the circuit and the physical I/O pins, and their use for I/O bound circuits is a standard design methodology.

The four multipliers, however, use only DSP blocks which contain their own internal registers. When virtual pins are used, the critical path of the circuit includes the delay of routing from the virtual pins to the DSP block input, and the delay of routing from the DSP block output to other virtual pins. The critical path delay is reduced significantly by connecting the physical I/O to the input/output registers of the DSP blocks directly, bypassing the virtual pins. This approach was taken in the following section for the four multipliers implemented using DSP blocks.

7. EXPERIMENTAL RESULTS

Figure 16 illustrates the critical path delays reported by Quartus II for the experiments described in the preceding section. Sections 7.1–7.4 discuss these results in detail for the four classes of benchmarks enumerated in Section 6.5. Sections 7.5 and 7.6 discuss trends in area utilization, and Section 7.7 presents results for pipelining.



■ *DSP/Untransformed*. Circuits are synthesized without transformation [Verma et al. 2008]; DSP blocks are used for the multiplier-based benchmarks. adpcm has no multipliers, but benefits from the transformations, so it does not use any DSP blocks.

▩ *Ternary*. Circuits are synthesized after transformation [Verma et al. 2008], which inhibits the use of DSP blocks. All multiplication and multi-input addition operations are synthesized on general FPGA logic using ternary adder trees.

▨ *GPC*. Same as above with all compressor trees synthesized on general FPGA logic using GPC mapping [Parandeh-Afshar et al. 2008b].

□ *FPCT*. Same as above with compressor trees synthesized on FPCTs.

⌶ 11% error estimate due to VEB methodology [Ho et al. 2006].

<u>Multipliers and Applications containing multipliers</u>

- Use DSP blocks for multiplication
- After transforming the DFG [Verma et al. 2008] DSP blocks cannot be used because the compressor trees within the multipliers have been merged with CPA operations.

<u>Naturally occuring multi-input adders and adpcm</u>

- These benchmarks contain no multipliers, and cannot use DSP blocks.
- fir3, fir6, and H.264 ME were written with all multi-input addition operations already clustered.
- Tranformations [Verma et al. 2008] are only applied to adpcm, which is the only one to be evaluated under *DSP/Untransformed*

Fig. 16. Results of synthesizing each benchmark three or four ways on the FPGA using four different methods: DSP/Untransformed, Ternary, GPC, and FPCT.

7.1 Critical Path Delay: Multipliers

The four multipliers are the benchmarks least favorable to the FPCT, as they are precisely the functionality for which DSP blocks are designed. Each half-DSP block in the Stratix II contains two 9×9-bit multipliers, and two half-DSP blocks can be combined to form a fast 18×18-bit multiplier. The increase in delay is marginal, just 0.06 ns, due to a adder tree that sums outputs from the four smaller multipliers.

$m9 \times 9$ is mapped onto a 9×9-bit multiplier in a half-DSP block, and also requires just a single FPCT. Due to the small bitwidth of this operation, the critical path delay using the DSP block is 0.015 ns faster than the FPCT, a statistically insignificant advantage.

$m10 \times 10$ is too large for 9×9-bit multiplication on half-DSP blocks, and therefore must be mapped onto an 18×18-bit multiplier. On the FPCT, $m10 \times 10$ uses the same number of CSlices as $m9 \times 9$, so its delay is identical. In this case, the FPCT retains an advantage is 0.045 ns, which is likewise statistically insignificant.

$m18 \times 18$ requires slightly more input bandwidth than a single FPCT provides. The input bandwidth of the FPCT (8×16-bit) is less than the input bandwidth of an 18×18-bit multiplier. $m18 \times 18$ requires three FPCTs in a vertical configuration, compared to a single DSP block. In this case, the DSP block reduces the critical path delay by 41% compared to the FPCT. The DSP block, however, has the advantage of hard logic that can efficiently combine smaller multipliers to form larger ones without using any soft FPGA logic or routing resources. If similar circuitry was made available to the FPCT, then the advantage enjoyed by the DSP block for $m18 \times 18$ is likely to reduce significantly.

$m20 \times 20$ has a bitwidth that is unfavorable to both FPCTs and DSP blocks. When using DSP blocks, $m20 \times 20$ is mapped to a $36 \times 36-$bit multiplier. On the other hand, it requires seven FPCTs, five of which form a vertical configuration along the critical path. Similar to $m18 \times 18$, the FPCT could benefit from a small amount of hard logic that could allow the combination of several FPCTs without using FPGA routing resources.

The impact of the confidence intervals in our measurement of critical path delay for FPCT, does not change the conclusions that we can draw for these benchmarks. As the DSP blocks are designed for fixed-bitwidth integer multiplication, the performance gap between DSP blocks and FPCTs will persist, with the delay measurements through the FPCT as much as 5.5% in excess of the numbers reported here.

Figure 17 shows the delay of each multiplier synthesized using each method, with the delays of Ternary, GPC, and FPCT decomposed into partial product generation (logic delay only) and the partial product reduction tree and final adder (including the cost of routing the partial products to the reduction tree). The delay for the DSP blocks is not decomposed in this fashion, as a DSP block

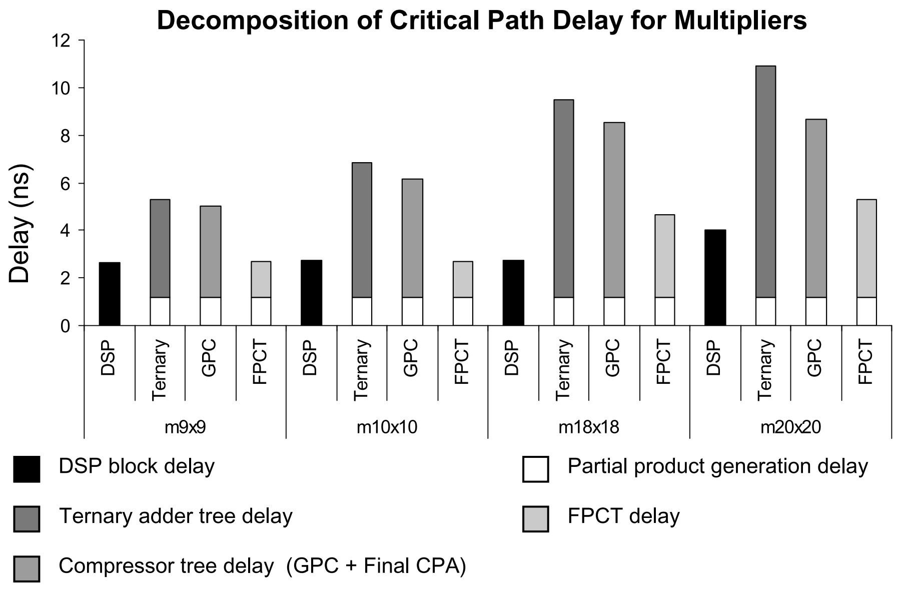


Fig. 17. Decomposition of the average critical path delay into partial product generation delay and adder/compressor tree delay (Ternary, GPC, FPCT), and comparison with the delay using DSP blocks.

can be viewed as an atomic element for comparison. Figure 17 shows that, for the most part, the delays through the FPCT are comparable to the delays through the DSP block; however, FPCT pays a heavy price for moving the partial product generator into the FPGA general logic. For the four multipliers, in order of increasing bitwidth, the percentage of the delay due to partial product generation is 45%, 45%, 26%, and 23%, respectively.

The gap between the delay of the partial product reduction tree through the FPCT and DSP block is largest for $m18 \times 18$, where the increase is 26%; on the other hand, for $m20 \times 20$, the difference is an insignificant 0.08 ns.

To summarize, these results illustrate two key points: (1) the DSP blocks will retain a significant advantage over the FPCT when the bitwidth of the operation best matches the former's capability; however, this advantage may dissipate if efficient hard logic structures are provided that permit the efficient concatenation of several FPCTs to form larger compressor trees, eliminating the cost of using the FPGA routing network for horizontal and vertical multi-FPCT configurations; and (2) the cost of partial product generation on the general logic of the FPCT is nonnegligible; to reduce this cost, future work should attempt to integrate partial product generation into the FPCT.

Lastly, we note that the results reported in Figure 17 show that neither Ternary nor GPC are competitive with either DSP blocks or the FPCT. These results have been included for completeness, but will not be analyzed in detail.

7.2 Critical Path Delay: Applications Containing Multipliers

*g*721, *hpoly*, and *video mixer* are larger applications that contain several multipliers as well as a variety of other CPA operations. The transformations of Verma et al. [2008] merge most of the multipliers' partial product reduction trees with other CPAs, forming larger compressor trees, but preventing the use of the DSP blocks afterwards.

For *g*721, FPCT breaks even with DSP/Untransformed, despite the use of Verma et al.'s transformations; the FPCT is 1% slower than DSP/Untransformed. The soft logic implementations, GPC and Ternary, are slower than both of these options. FPCT achieved a 6.4% reduction in critical path delay compared to GPC. The confidence interval suggests that the use of the FPCT could cause the critical path of the circuit to exceed that of GPC. This seems highly unlikely, and suggests that the 11% confidence interval we have assumed probably overestimates the actual error in our measurements.

hpoly benefits considerably from the transformations, as the critical path delay of Ternary is 7.7% faster than the critical path delay when DSP blocks are used. GPC and FPCT are increasingly effective, achieving respective reductions in critical path delay, relative to DSP/Untransformed, of 20% and 34%. Compared to GPC, FPCT reduced the critical path delay by 18%. *hpoly* quantitatively demonstrates that even multiplier-based applications can benefit from hardware blocks that provide more general acceleration for carry-save arithmetic than the DSP blocks.

For *video mixer*, after applying the transformations, Ternary exhibits a noticeable slowdown compared to DSP/Untransformed; however, GPC eliminates most of this slowdown, as its critical path is less than 1% slower than DSP/Untransformed. FPCT, in contrast, achieves a speedup of 32% compared to DSP/Untransformed, and 33% compared to GPC. *Video mixer* illustrates the fact that the FPCT can accelerate carry-save-based computations that the DSP blocks cannot.

To summarize, *hpoly* and *video mixer* illustrate that, in the most general case, DSP blocks do not sufficiently accelerate carry-save addition in its most general form, even when a good portion of the compressor trees originate from multipliers. The strength of the FPCT, in contrast, is that it does not impose restrictions on the origins and transformations of the compressor trees that it accelerates. On the other hand, as discussed in Section 7.1, the cost of moving the partial product generator onto the general FPGA logic is a source of concern in the more general case, despite the fact that these three applications do not suffer from the costs of doing so.

7.3 Critical Path Delay: Applications Naturally Containing Multi-input Adders

fir3, *fir6*, and *H.264 ME* all contain large multi-input addition operations that do not occur in the context of multipliers, and cannot use DSP blocks. As all CPAs are already clustered together, the transformations of Verma et al. cannot improve these circuits. In a sense, they have been designed "optimally" by hand, as all opportunities to use carry-save arithmetic are exposed a priori. For all three of these benchmarks, Ternary has the longest critical path,

Table III. Resource Utilization for DSP/Untransformed and FPCT for the Four Multipliers and Three Larger Benchmark Circuits Containing Multipliers

	DSP/Untransformed		FPCT	
Benchmark	ALMs	Half-DSP Blocks	ALMS	FPCTs
m9x9	0	1	47	1
m10x10	0	4	50	1
m18x18	0	4	165	3
m20x20	0	16	200	7
g721	111	4	156	3
hpoly	32	16	304	3
video mixer	213	300	40	78

followed by GPC and then FPCT. Relative to GPC, the speedups achieved by FPCT are 22%, 49%, and 36%.

These results demonstrate that applications exist for which multi-input occurs naturally, and that the transformations of Verma et al. are not always required to expose opportunities to exploit carry-save addition. Moreover, since the compressor trees in these applications do not originate from parallel multipliers, the DSP blocks cannot be used. Therefore, compared to the DSP block, the FPCT can accelerate a wider range of applications, with the caveat that multiplication of certain bitwidths that are most favorable to the DSP blocks will experience some performance impediment.

7.4 Critical Path Delay: adpcm

adpcm is similar to the benchmarks discussed in Section 7.3; the primary difference is that its native implementation in C code does not cluster the three CPAs together, as shown in Figure 1; thus, *adpcm* is evaluated both with and without the transformations. As the clustered multi-input addition operation has only 4 inputs, the opportunities for acceleration are limited. Compared to the baseline (DSP/Untransformed), the reductions in critical path delay achieved by Ternary, GPC, and FPCT are 18%, 33%, and 49%, respectively. Once again, *adpcm* is a benchmark that the FPCT can accelerate, whereas, DSP blocks cannot. The confidence interval indicates that FPCT will achieve a speedup over GPC in all cases, but that it may, in the worst case, be marginal.

To conclude, *adpcm* yields similar conclusions as given in the preceding section, but also shows that Verma et al.'s transformations are useful for benchmarks without multipliers.

7.5 Area Comparison: DSP Blocks vs. the FPCT

Table III compares the resource utilization for DSP/Untransformed and FPCT for the seven circuits that contained multipliers, and could exploit DSP blocks. Based on Section 6.2, we assume that the area of a half-DSP block is equivalent to the area of an FPCT. Table III shows that the number of FPCTs used per benchmark never exceeds the number of half-DSP blocks; however, FPCT requires more ALMs than DSP/Untransformed DSP, since partial product generation is done on the FPGA general logic. Based on the discussion in Section 6.3,

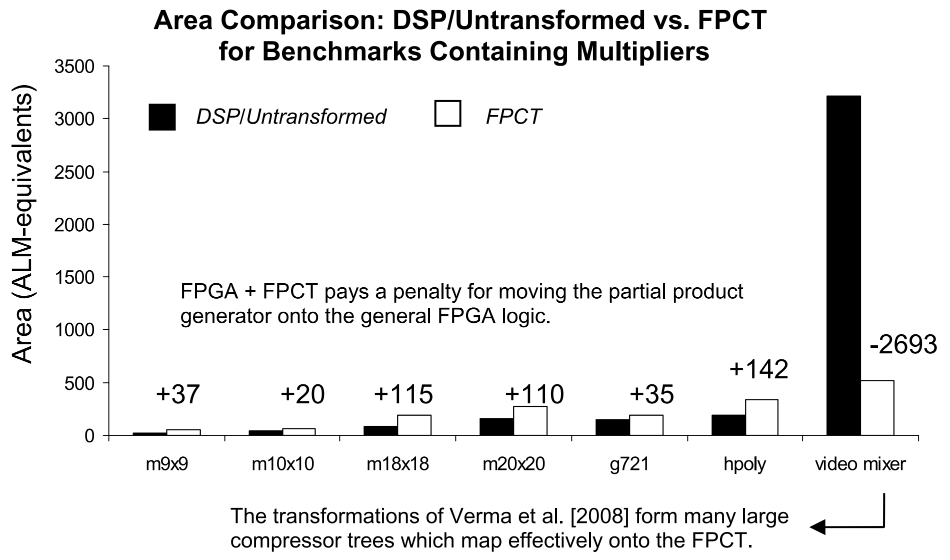


Fig. 18. Area of FPGA + DSP and FPGA + FPCT for the multiplier-based benchmarks. The area is expressed in terms of ALM equivalents.

we assume that the area of a half-DSP block and an ALM are equivalent to the area of ten ALMs; we call these units *ALM-equivalents*.

Figure 18 compares the area of DSP/Untransformed and FPCT, expressed as ALM-equivalents, for the multiplier-based applications and multipliers. For $g721$, *hpoly*, $m9 \times 9$, $m10 \times 10$, $m18 \times 18$, and $m20 \times 20$, the use of FPCTs increases the area of the circuit because the partial product generators are synthesized on the FPGA general logic. *video mixer*, in contrast, benefits from the transformations of Verma et al. [2008], which merge many adders and multipliers to form large compressor trees that map efficiently onto FPCTs quite well, but cannot map onto DSP blocks, considerably reducing area.

Table IV and Figure 19, respectively, compare the resource utilization and area (ALM-equivalents), for Ternary, GPC, and FPCT. Table IV expresses the compressor tree requirements in terms of ALMs for Ternary and GPC and FPCTs for FPCT. For all benchmarks the noncompressor tree logic is identical, and FPCT had the smallest area requirement. A CSlice is approximately equal to an ALM in size; however, an ALM has six inputs, while a CSlice has sixteen.

Under Ternary, one ALM compresses six input bits down to two; under GPC, two ALMs are required to realize a GPC with six inputs and three or four outputs. FPCT reduces area compared to Ternary and GPC because each CSlice has a higher bandwidth than an ALM; this explains FPCT's advantage in terms of area utilization.

7.6 Results for Pipelined Compressor Trees

The results in the preceding sections used single-cycle compressor trees for all applications. This section summarizes an experiment in which each

Table IV. Resource Utilization for Ternary, GPC, and FPCT for All Benchmarks

	Non-Compressor Tree Logic (ALMs)	Adder/Compressor Tree Logic		
		Ternary (ALMs)	GPC (ALMs)	FPCT (FPCTs)
m9×9	47	36	44	1
m10×10	50	52	51	1
m18×18	165	162	190	3
m20×20	200	220	229	7
g721	156	116	134	3
hpoly	304	190	163	3
video mixer	40	543	650	48
fir3	0	166	161	6
fir6	0	320	329	8
H264 ME	423	160	224	4
adpcm	20	20	24	1

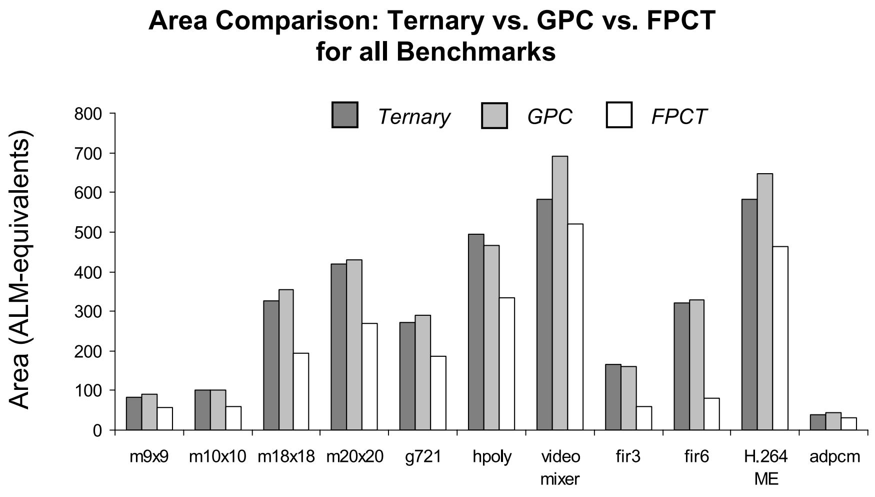


Fig. 19. Area (ALM-equivalents) for Ternary, GPC, and FPCT for all benchmarks.

multi-FPCT configuration was pipelined. The purpose of this section is to illustrate that the FPCT can offer high clock speeds, just like DSP blocks, for throughput-oriented applications; the latency, that is, the number of pipeline stages, is not a primary concern here.

Only the compressor tree portions of each benchmark are synthesized. With the exception of *video mixer*, all benchmarks contain one compressor tree. Each compressor tree in *video mixer* is treated separately; their names prefaced by *vm.*, for example, *vm.add2I*.

Table V shows the results of the experiment. For each benchmark, Table V lists the number of FPCTs used along with the latency, operating frequency, and percentage of critical path delay due to routing between pipeline stages. The clock frequencies ranged from 432 MHz to 707 MHz, which are competitive with 550 MHz DSP blocks.

For complete applications, the maximum clock frequency may be less than what is reported in Table V for these compressor trees, due to chaining

Table V. Compressor Trees Synthesized Using Pipelined Multi-FPCT Configurations

Benchmark	FPCTs	Latency	Frequency (MHz)	Delay percentage due to Routing
g72x	3	3	664	1%
hpoly	3	3	664	1%
vm.Add2I	8	5	493	37%
vm.Add2Q	2	2	665	1%
vm.Add2Y	6	5	494	52%
vm.adt2	15	7	434	54%
vm.adt7	9	5	485	39%
vm.RQGQBQB	4	3	450	55%
vm.RYGYBYB	4	3	450	53%
adpcm	1	1	707	0
fir3	6	5	431	56%
fir6	8	5	446	55%
H.264 ME	1	1	672	0
m9x9	1	1	681	0
m10x10	1	1	681	0
m18x18	3	3	657	0
m20x20	7	5	476	41%

The number of FPCTs, latency (number of pipeline stages), maximum clock frequency, and the percentage of the critical path delay due to routing are shown.

noncompressor tree logic along with compressor trees in a single cycle, and the possibility that the compressor trees may not be on the critical path. In several cases, the contribution of the FPGA routing network to overall critical path delay was above 50%.

We do not compare pipelined FPCTs with DSP blocks because the latter's partial product reduction trees are not pipelined. Ternary and GPC are likely to achieve even higher clock rates but with longer latencies than the DSP blocks. If aggressively pipelined, each ternary adder or GPC output would be written to a register. The granularity of pipelining after each level of LUTs in the compressor or adder tree is much finer than pipelining each FPCT output, even when accounting for routing delays.

Lastly, we compared the ability of the ASAP scheduling heuristic to reduce the number of ALMs required to realize external pipeline registers. Each compressor tree was synthesized using ASAP and ALAP scheduling. Figure 20 quantifies the savings achievable when ASAP scheduling is used in lieu of ALAP scheduling.

In four cases, *apdcm*, $m9\times9$, $m10\times10$, and *H.264 ME*, only one FPCT was required so there was no pipelining. In four other cases, *vm.add2Q*, $g72x$, *hpoly*, and $m18 \times 18$, all FPCTs were on the critical path, so the ASAP and ALAP schedules were identical. In four other cases, *vm.RQGQBQB*, *vm.RYGYBYB*, *fir3*, and $m20 \times 20$, ALAP required a handful more ALMs than ASAP scheduling for pipeline registers (2, 3, 8, and 9, respectively).

For the remaining benchmarks, the number of ALMs instantiated for external pipeline registers by the ALAP heuristic was significantly greater than the number instantiated by the ASAP heuristic, the maximum difference being 315 ALMs for *adt2*. On average, ALAP scheduling allocated 18.3% more ALMs to external pipeline registers than ASAP scheduling.

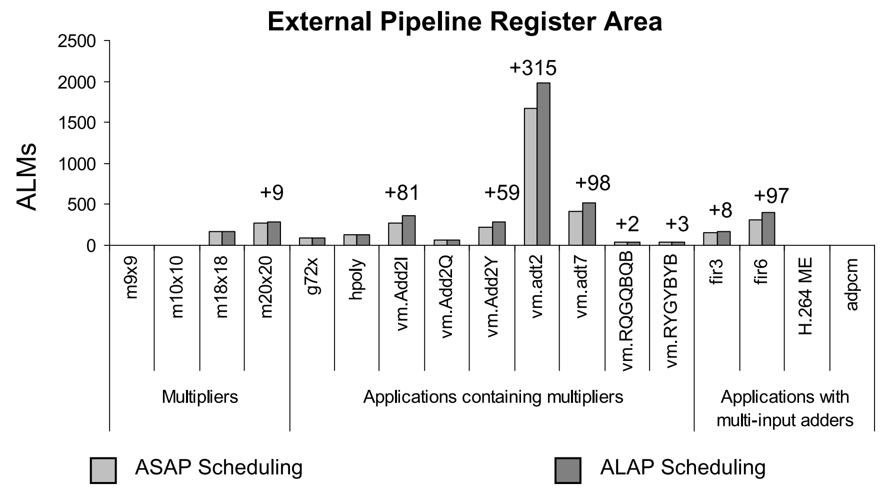


Fig. 20. The number of ALMs required to instantiate external pipeline registers when ASAP and ALAP scheduling are used to generate pipelined multi-FPCT configurations. The difference, when nonzero, is shown above each benchmark, for example, ALAP required 81 more ALMs than ASAP for *vm.add2I*).

8. CONCLUSION AND FUTURE WORK

The FPCT is a radical departure from the traditional DSP blocks that currently enhance the arithmetic functionality of FPGAs. As discussed in our first attempt to architect an FPCT [Brisk et al. 2007], the motivation for the FPCT originates from a set of data flow transformations [Verma et al. 2008] that improve arithmetic circuits for ASIC synthesis by maximizing the use of carry-save addition whenever possible. Our intention since then has been to study methods to allow FPGAs to benefit from these transformations as well; the FPCT is the primary result of this study.

For most of the benchmarks studied in this article, the FPCT, in conjunction with the transformations, outperformed the DSP blocks; however, the DSP blocks retained their advantage for multiplication operations whose partial product reduction trees were not modified by the transformations. The loss appears to originate from two sources: the movement of the partial product generator onto soft FPGA logic, and the replacement of an ASIC implementation of a fixed function with a reconfigurable alternative.

The cost of the latter is unavoidable, but mostly insignificant, as illustrated by Figure 17; however, the cost of the former is nonnegligible for two reasons. Firstly, the delay of implementing partial product generators with a layer of LUTs is greater than using AND gates; however, a more pressing concern is the congestion in the routing network that may result. Multiplying two m- and n-bit integers produces mn partial product bits. When an FPCT replaces a DSP block, each multiplication operation increases the $m + n$ bits that must be routed to the DSP block with mn bits that are routed to an FPCT. This is of particular importance for system-level benchmarks that contain hundreds of multipliers; far more than the benchmarks evaluated in this study. Future work will attempt to tame the increased pressure on the routing network using novel hardware and/or improved mapping techniques.

REFERENCES

ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming*. 177–189.

ALTERA CORPORATION. 2006. Stratix II performance and logic efficiency analysis. White paper. September. http://www.altera.com/.

ALTERA CORPORATION. 2008a. Stratix II device handbook. http://www.altera.com/.

ALTERA CORPORATION. 2008b. Stratix III device handbook. http://www.altera.com/.

ALTERA CORPORATION. 2008c. Stratix IV device handbook. http://www.altera.com/.

BEUCHAT, J.-L. AND TISSERAND, A. 2002. Small multiplier-based multiplication and division operators for Virtex-II devices. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*. 513–522.

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement, and routing tool for FPGA research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. 213–222.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep Submicron FPGAs*. Kluwer Academic, Norwell, MA.

BRISK, P., VERMA, A. K., IENNE, P., AND PARANDEH-AFSHAR, H. 2007. Enhancing FPGA performance for arithmetic circuits. In *Proceedings of the 44th Design Automation Conference*. 404–409.

CEVRERO, A., ATHANASOPOULOS, P., PARANDEH-AFSHAR, H., VERMA, A. K., BRISK, P., ET AL. 2008. Architectural improvements for field programmable counter arrays: Enabling efficient synthesis of fast compressor trees on FPGAs. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*. 181–190.

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.-I 53*, 578–593.

CHEREPACHA, D. AND LEWIS, D. 1996. DP-FPGA: An FPGA architecture optimizated for datapaths. *VLSI Des. 4*, 329–343.

COSOROABA, A. AND RIVOALLON, F. 2006. Achieving higher system performance with the Virtex-5 family of FPGAs. White paper: Xilinx Corporation. July. http://www.xilinx.com/.

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

FREDERICK, M. T. AND SOMANI, A. K. 2006. Multi-bit carry chains for high performance reconfigurable fabrics. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. 1–6.

HAUCK, S., HOSLER, M. M., AND FRY, T. W. 2000. High-performance carry chains for FPGAs. *IEEE Trans. VLSI Syst. 8*, 138–147.

HO, C. H., LEONG, P. H. W., LUK, W., WILTON, S. J. E., AND LOPEZ-BUEDO, S. 2006. Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*. 35–44.

KAVIANI, A., VRANISEC, D., AND BROWN, S. 1998. Computational field programmable architecture. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 261–264.

KUON, I. AND ROSE, J. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Comput.-Aided Des. 26*, 203–215.

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 30th International Symposium on Microarchitecture*. 330–335.

LEIJTEN-NOWAK, K. AND VAN MEERBERGEN, J. L. 2003. An FPGA architecture with enhanced datapath functionality. In *Proceedings of the 11th International Symposium on FPGAs*. 195–204.

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.

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.

PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. 2008a. A novel FPGA logic block for improved arithmetic performance. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*. 171–180.

PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. 2008b. 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. 2008c. 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. Scalable and low cost design approach for variable block size motion estimation. In *Proceedings of the International Symposium on VLSI Design Automation and Test*.

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.

SRIRAM, S., BROWN, K., DEFOSSEUX, R., MOERMAN, F., PAVIOT, O., SUNDARARAJAN, V., AND GATHERER, A. 2005. A 64 channel programmable receiver chip for 3G wireless infrastructure. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 59–62.

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.

VERMA, A. K., BRISK, P., AND IENNE, P. 2008. Data-Flow transformations to maximize 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.

XILINX CORPORATION. 2008a. Virtex-5 FPGA XtremeDSP design considerations. http://www.xilinx.com/.

XILINX CORPORATION. 2008b. Virtex-5 user guide. http://www.xilinx.com/.

ZUCHOWSKI, P. S., REYNOLDS, C. B., GRUPP, R. J., DAVIS, S. G., CREMEN, B., AND TROXEL, B. 2002. A hybrid ASIC and FPGA architecture. In *Proceedings of the International Conference on Computer-Aided Design*. 187–194.

Received June 2008; revised January 2009; accepted February 2009