Improving FPGA Performance for Carry-Save Arithmetic

Hadi Parandeh-Afshar, Ajay Kumar Verma, Philip Brisk, and Paolo Ienne

***Abstract*—The selective use of carry-save arithmetic, where appropriate, can accelerate a variety of arithmetic-dominated circuits. Carry-save arithmetic occurs naturally in a variety of DSP applications, and further opportunities to exploit it can be exposed through systematic data flow transformations that can be applied by a hardware compiler. Field-programmable gate arrays (FPGAs), however, are not particularly well suited to carry-save arithmetic. To address this concern, we introduce the "field programmable counter array" (FPCA), an accelerator for carry-save arithmetic intended for integration into an FPGA as an alternative to DSP blocks. In addition to multiplication and multiply accumulation, the FPCA can accelerate more general carry-save operations, such as multi-input addition (e.g., add $k > 2$ integers) and multipliers that have been fused with other adders. Our experiments show that the FPCA accelerates a wider variety of applications than DSP blocks and improves performance, area utilization, and energy consumption compared with soft FPGA logic.**

***Index Terms*—Carry-save arithmetic, field-programmable gate array (FPGA), generalized parallel counter (GPC).**

I. Introduction

FIELD-PROGRAMMABLE-GATE-ARRAY (FPGA) performance is lacking for arithmetic circuits. Generally, arithmetic circuits do not map well onto lookup tables (LUTs), the primary building block for general logic in FPGAs. To address this concern, FPGAs offer two solutions: First, LUTs are now tightly integrated with fast carry chains that perform efficient carry-propagate addition; second, FPGAs contain DSP blocks that perform multiplication and multiply accumulation (MAC). Although an improvement over LUTs alone, these enhancements lack generality; specifically, they cannot effectively accelerate carry-save arithmetic.

Carry-save arithmetic is a technique to add sets of $k > 2$ numbers that eliminates much of the carry propagation that would otherwise occur. Carry-save arithmetic has been the method of choice for partial-product reduction in parallel multipliers for more than 40 years [1], [2]. More recently, Verma *et al.* [3] developed a set of arithmetic-oriented data flow transformations that can be applied to a computation in order to maximize the use of carry-save arithmetic. These transformations systematically reorder the operations in a circuit in order to cluster disparate adders together and to merge adders with the partial-product-reduction trees of parallel multipliers. Each cluster of adders is then replaced with a *compressor tree*, i.e., a circuit that reduces n integers, $A_0, A_1, \ldots, A_{n-1}$, down to two, S (sum) and C (carry), such that

$$S + C = \sum_{i=0}^{n-1} A_i. \tag{1}$$

A carry-propagate adder (CPA), i.e., a two-input adder, then performs the final addition, $S + C$, to compute the result. Aside from the transformations of Verma *et al.* [3], compressor trees occur naturally in a variety of applications [4]–[8].

The arithmetic capabilities of FPGAs are not well attuned to the needs of carry-save arithmetic. Programmable LUTs have been augmented with fast carry chains that are good building blocks for CPAs but cannot be used for carry-save arithmetic. The fastest methods to synthesize compressor trees on FPGA general logic [9], [10] do not use the carry chains except for the final CPA.

FPGAs also integrate DSP blocks, which perform integer multiplication and MAC. Although useful, DSP blocks cannot accelerate multi-input addition; likewise, when the transformations of Verma *et al.* merge multipliers with adders, the resulting operation can no longer map onto a DSP block. That being said, certain multiplication operations whose bitwidths do not match up well with the bitwidths of the DSP blocks are faster when performed on the general logic of an FPGA [11].

This paper advocates the use of a field programmable counter array (FPCA) for carry-save arithmetic on FPGAs. The FPCA is a programmable accelerator that can be integrated into an FPGA as an alternative to DSP blocks. An early FPCA, introduced by Brisk *et al.* [12], is a lattice of $m : n$ counters. An $m : n$ counter is a circuit that takes m input bits, counts the number of them that are set to 1, and outputs the result, a value in the range of $[0, m]$, as an n-bit unsigned binary number. The number of output bits is

$$n = \lceil \log_2(m + 1) \rceil . \tag{2}$$

The FPCA described in this paper, in contrast, is built using generalized parallel counters (GPCs) [13]–[15], an extended type of counter that can sum bits having different input ranks; GPCs, which will be defined formally in Section III, are built using $m : n$ counters as building blocks.

Our experiments compare the FPCA with DSP blocks for multiplication-dominated circuits and with the best methods to synthesize compressor trees on general FPGA logic for other circuits that feature carry-save arithmetic. As the DSP blocks are fixed-bitwidth multipliers/MACs, they perform better than

Manuscript received May 27, 2008; revised September 26, 2008. First published June 16, 2009; current version published March 24, 2010.

The authors are with the Processor Architecture Laboratory, School of Computer and Communications Sciences, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland (e-mail: hparande@gmail.com; ajaykumar.verma@epfl.ch; philip.brisk@epfl.ch; paolo.ienne@epfl.ch).

Digital Object Identifier 10.1109/TVLSI.2009.2014380

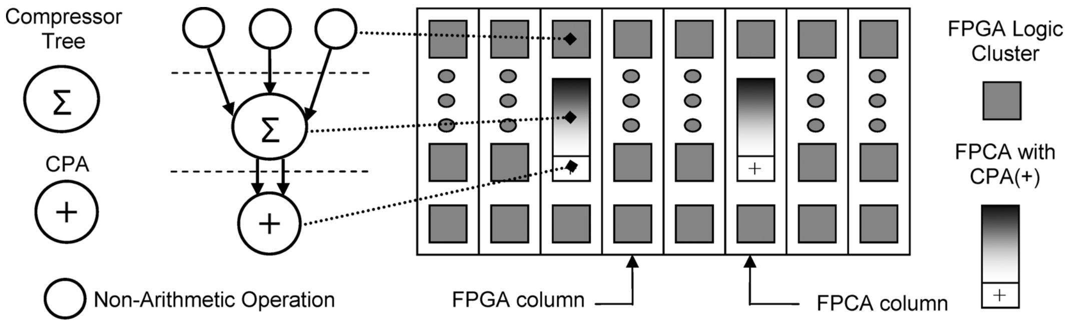


Fig. 1. Illustration of the methodology underlying the proposed reconfigurable lattice. Using arithmetic transformations [3], a circuit/flow graph is transformed to expose one (or more) compressor trees. The compressor tree is mapped onto a new reconfigurable lattice (FPCA) that is integrated into an FPGA. The final addition is mapped onto a dedicated adder (shown here, integrated into the new lattice but could easily be implemented using a carry chain instead). The remaining nonarithmetic operations in the circuit/flow graph are mapped onto the FPGA logic blocks.

the FPCA for those operations and bitwidths; however, the FPCA retains an advantage when bitwidth mismatches occur. The FPCA also benefits from the transformations of Verma *et al.* [3], whereas the DSP blocks do not. The FPCA offers advantages over DSP blocks and FPGA logic in terms of critical path delay, area utilization, and energy consumption.

In our experiments, we considered GPCs built from four parallel counter sizes: 8:4, 12:4, 16:5, and 20:5; although no counter size was uniformly better than the others across all benchmarks, our results suggest that increasing the counter size beyond 12:4 yields diminishing returns. We conclude that GPCs built from 12:4 counters are the ideal choice for our specific set of benchmarks.

A. Illustrative Example

Fig. 1 shows our approach. A circuit transformed as described previously is partitioned into a (set of) compressor tree(s) with corresponding CPA(s) and a set of nonarithmetic operations. The compressor tree is mapped onto an FPCA, which is embedded within a larger FPGA. Fig. 1 assumes that a dedicated CPA is integrated into the FPCA; alternatively, the carry chains in the logic-block structure of the FPGA could be used to perform the final CPA. The nonarithmetic portions of the circuit are mapped onto the FPGA. Following the lead of Xilinx and Altera, the FPGA shown in Fig. 1 is organized into columns. Each column contains a set of logic clusters [e.g., the Altera Logic Array Block (LAB)], which contain several logic blocks [e.g., the Altera Adaptive Logic Module (ALM)] connected by local routing. A global routing network connects the different logic clusters. Due to the column structure, the horizontal and vertical routing channels are nonuniform.

B. Paper Organization

The paper is organized as follows. Section II summarizes related work, Section III introduces GPCs, Section IV presents the FPCA architecture, Sections V and VI present the experimental framework and results, and lastly, Section VII concludes the paper.

II. Related Work

A. Commercial FPGA Architectures and Mapping

This section summarizes the arithmetic features in the *Altera Stratix III* [16] and *Xilinx Virtex-5* [17] FPGAs, both of which are high-end FPGAs realized in 65-nm CMOS technology. The logic architectures of both of these FPGAs feature six-input LUTs with carry chains that perform efficient carry-propagate addition without using the routing network. The *Stratix III* carry chain is a ripple-carry adder; the *Virtex-5* carry chain includes an XOR gate and a multiplexor (mux) which enable carry-lookahead addition.

Stratix II introduced a method to combine the LUTs with the carry chain to perform ternary (three-input) addition, which remained in place for the *Stratix III*; the *Virtex-5* similarly supports ternary addition.

Due to the peculiar nature of FPGA architectures, it has long been thought that multi-input addition is best realized using trees of adders rather than compressor trees. The use of ternary adders rather than binary (two-input) adders could reduce the height of the trees, thereby reducing delay and/or pipeline depth. Parandeh-Afshar *et al.* [9], [10], however, showed that compressor trees could be synthesized on FPGAs using GPCs (see Section III), significantly reducing the delay compared to ternary adder trees. Experimentally, this paper finds that the FPCA is faster than both of these alternatives.

B. FPGA Enhancements to Improve Arithmetic Performance

Numerous enhancements for FPGAs have been proposed in the past, particularly to improve arithmetic performance. For example, several researchers have proposed hard IP cores: application-specific integrated circuit (ASIC) components that implement common operations that are embedded into the FPGA. The most prevalent of these IP cores include block memories, DSP/MAC blocks [18], [19], standard I/O interfaces [19], crossbars [20], shifters [21], and floating-point units [21]. Kastner *et al.* [22] developed techniques to examine a set of applications to find good domain-specific IP core candidates.

Although the FPCA is similar in principle to the IP cores described previously, it is not completely hard: It is programmable and has its own routing network. Although it is intended to implement just one class of circuits—compressor trees, the FPCA is flexible and is not fixed to a specific bitwidth; this distinguishes the FPCA, for example, from the hard multipliers whose bitwidths are fixed. Kuon and Rose [11] have noted that fixed bitwidth multiplication has some limitations, e.g., it is inefficient to implement 5×5-b multiplication on a 9×9-b multiplier contained in a DSP block.

Cevrero *et al.* [23] recently proposed an alternative FPCA architecture. Theirs is radically different than the one described here; the most important distinguishing feature is that it uses direct programmable connections but does not employ a global routing network; as such, it offers less flexibility than the architecture proposed here, but with the potential of reduced delay, area, and power consumption due to the absence of global routing. Future work will compare and contrast these two architectures to better understand the differences between them.

C. Carry Chains

Also notable but not directly related to this work are the fast carry chains [24]–[29]: These are used to implement efficient carry-propagate addition within FPGA logic cells. If an FPCA is present, these carry chains can be used to perform the final addition if a hard IP core implementation of a CPA is not present.

Parandeh-Afshar *et al.* [29] developed a carry chain that allows a logic cell to be configured as a 6:2 compressor, a well-known building block for compressor trees. The FPCA, however, is much more powerful, as its logic cells contain larger GPCs (e.g., with up to 20 inputs). The use of larger and more flexible components reduces the number of logic levels in the compressor tree as well as pressure on the routing network: This is favorable from the perspective of delay, area utilization, and power consumption.

D. Programmable Arrays of Arithmetic Primitives

The FPCA is a homogeneous array of arithmetic primitives connected by a routing network. Many principally similar arithmetic arrays have been proposed in the past, and this similarity is acknowledged. The main difference is that the FPCA is limited in its scope of application (solely compressor trees) and is intended for integration into a larger FPGA, whereas the arrays discussed in this paper are stand-alone devices.

Parhami [30], for example, built an array of bit-serial additive multipliers and used a data-driven control scheme. The advantage of bit-serial arithmetic is that it reduces the wiring requirement for an FPGA: This is significant because wiring can consume up to 70% of on-chip area. Although somewhat beyond the scope of this paper, bit-serial routing networks are an active area of research that is beginning to emerge [31]; the applications for such a device, however, must be able to tolerate high latencies, and it is not immediately clear which applications easily fall into this category.

The reconfigurable arithmetic FPGA (RA-FPGA) [32] is an arithmetic array partitioned into three regions: 1) two's complement addition; 2) sign/magnitude conversion to two's complement, and vice versa; and 3) multiplication and division. Traditional FPGA-style logic is also included in order to implement control and general-purpose logic. In principle, such a device could use an FPCA to perform multiplication; however, no RA-FPGA has been produced commercially to the best of our knowledge.

The *CHESS* reconfigurable array [33], developed at *HP Labs*, is an array of 4-b arithmetic logic units (ALUs), connected by a bus-based FPGA-style routing network. Each ALU supports 16 arithmetic and logical operations (e.g., ADD, SUB, XOR), along with selection and comparison tests. Neighboring ALUs can be chained together (e.g., to perform 8-b addition), and spatial parallelism is abundant. As the ALU does not support primitives for multiplication or multioperand addition, the inclusion of an FPCA into the array is certainly plausible.

Several configurable arrays of floating-point units have also been proposed. In 1988, Fiske and Dally [34] introduced the Reconfigurable Arithmetic Processor (RAP), which contains 64 floating units connected by a switching network. More recently, *Intel's Teraflops* processor [35] connected 80 floating-point MAC units using a high-speed network on chip. Although floating-point units contain integer multipliers (and, hence, compressor trees), it does not appear that there would be any room to incorporate an FPCA into such a chip, because the floating-point units themselves have fixed bitwidths in accordance with IEEE standards.

III. GPCs

A. Overview

Let $B = (b_{n-1}, b_{n-2}, \ldots, b_1, b_0)$ be an n-bit binary number, where each b_j, $0 \leq j \leq n-1$ is a bit. Let b_0 be the least bit and b_{n-1} be the most significant bit. The subscript of a bit, in this case, is called the $rank$ of the bit. Each bit b_j has rank j and contributes a value of $b_j 2^j$ to the total quantity represented by the binary integer.

An $m : n$ counter, as described in the preceding section, assumes that all bits have the same rank when it computes their sum. If all input bits have rank j, then the output of the $m : n$ counter is a set of n bits having ranks $j, j+1, \ldots, j+n-1$.

A GPC [13]–[15] is a type of counter that counts bits having different ranks. In fact, an $m : n$ counter can implement a GPC, if desired: A bit of rank r must be connected to precisely 2^r inputs of the $m : n$ counter. Of course, other methods to build GPCs also exist.

A GPC is defined as a tuple $P = (K_{N-2}, K_{N-1}, \ldots, K_1, K_0; n)$, where K_j is the number of input bits of rank j to sum and n is the number of output bits; the input bits of each rank are independent. For example, a (5, 3; 4) GPC can count up to 5 b of rank-1 and 3 b of rank-0; the maximum output value is 3; therefore, four output bits are required.

Here, we fix the number of input and output bits to be positive constants m and n. Given m and n, there is a family of GPCs that satisfy these I/O constraints. Clearly, the number of input bits cannot exceed m

$$\sum_{j=0}^{N-2} K_j \leq m. \tag{3}$$

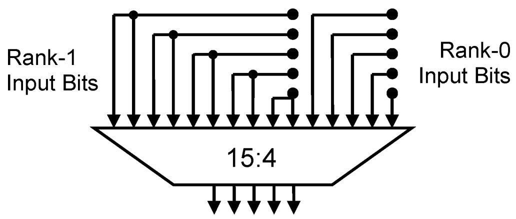


Fig. 2. 15:4 counter can implement a (5, 5; 4) GPC.

Likewise, the maximum allowable output value, which occurs when all input bits are "1," cannot exceed the maximum integer value that can be expressed with n output bits

$$0 \leq \sum_{j=0}^{N-2} K_j 2^j \leq 2^n - 1. \tag{4}$$

For a given m and n, there is a family of GPCs that satisfy these I/O constraints. As an example, take $m = 10$ and $n = 5$. One GPC in this family is a (5, 5; 4) GPC (see Fig. 2).

This GPC has five input bits of rank-0 and five of rank-1. The maximum value that can be counted is $5 \times 2^1 + 5 \times 2^0 = 15$; four output bits are required to represent a value in the range [0, 15]; clearly, any $(K_1, K_0; 4)$ GPC suffices, under the assumption that $0 \leq K_1$ and $K_0 \leq 5$ as well.

At the same time, a (4, 6; 4) GPC is also a member of this family, as it produces an output value in the range [0, 14], as is a (6, 3; 4) GPC, etc. An $m : n$ counter is also a degenerate case of a GPC; in this case, a (0, 10; 4) GPC.

B. *Configurable GPCs*

For the FPCA, one fixed GPC does not suffice, because we desire more flexibility. Instead, we build a *configurable* GPC using an $m : n$ counter with a layer of muxes placed on its input. This *configuration layer*, which is described in detail in Section III-D, allows the configurable GPC to implement a wide variety of GPCs; the user selects the desired GPC to implement and programs the configuration layer accordingly. For example, a programmable GPC with $m = 15$ and $n = 4$ should be able to implement the functionality of both a 15:4 counter and a (5, 5; 4) GPC, among others.

The $rank$ of a GPC is the minimum rank among all of its input bits. Now, suppose that the minimum rank of an input bit to a given GPC is i and that the GPC must add two (or more) bits of ranks i and j, such that $j > i$. Each input bit of rank i is connected to one input of the GPC, while each input bit of rank j is connected to $2^{(j-i)}$ inputs. Fig. 2, for example, satisfies this property.

Fig. 3 shows the design of an m-input n-output GPC built from an m-input m-output configuration layer followed by an $m : n$ counter. Each input of the $m : n$ counter (output of the configuration layer) is connected to two GPC inputs and is controlled by two configuration bits. The configuration bit on the left selects one of two GPC inputs that are connected to a mux; the configuration bit on the right drives the counter input to 0 if it is not set, which allows the $m : n$ counter to implement any $m' : n'$ counter for $m' \leq m$ and $n' \leq n$; in this case, the $m' : n'$ counter is called a *subcounter*. For example, a 7:3 counter has five subcounters: 6:3, 5:3, 4:3, 3:2, and 2:2 counters. The definition of subcounters easily extends to GPCs as well.

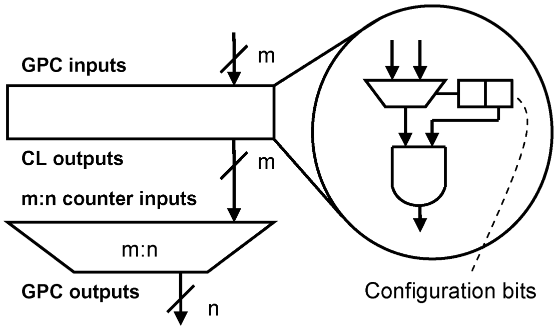


Fig. 3. Architecture of an m-input programmable GPC.

C. *Primitive, Covering, and Reasonable GPCs*

This section specifies precisely which m-input n-output GPCs should be implemented by the programmable GPC with m inputs, n outputs, and an $m : n$ counter at its core.

A *primitive* GPC is one that satisfies the I/O constraints.

A *covering* GPC is a primitive GPC that is not a sub-GPC of another primitive GPC. Referring to Fig. 2, the (5, 5; 4) GPC is a covering GPC. If the number of rank-1 inputs is increased to six, then, there are only three rank-0 input ports remaining, i.e., $6 \times 2^1 + 3 \times 2^0 = 15$. Hence, a (6, 3; 4) GPC is also a covering GPC. A (5, 4; 4) GPC, in contrast, is *not* a covering GPC, because the (5, 5; 4) GPC can implement its functionality by driving one of the rank-0 inputs to zero.

To achieve *universal coverage*, a configurable GPC only needs to implement the functionality of the covering GPCs that have m inputs and n outputs. Of course, there is no formal mandate that a configurable GPC provide universal coverage; those that do not simply have limited flexibility compared with those that do.

We have identified two classes of *unreasonable* GPCs, meaning that we can find no rational justification for using them; this is not, however, a formal definition. When designing a configurable GPC, there is no need to add support for unreasonable GPCs, even if they are covering GPCs.

A GPC that has no rank-0 input bits, i.e., $K_0 = 0$, is unreasonable. For example, consider a (7, 0; 4) GPC. The rank-0 output will always be 0. The rank-1, -2, and -3 outputs can be computed by a 7:3 counter. If the rank of this GPC is i, it suffices to replace it with a 7:3 counter of rank $i + 1$ instead. Thus, a configurable GPC need not support this type of GPC.

Similarly, a GPC that has one rank-0 input bit, i.e., $K_0 = 1$, is unreasonable. This bit determines whether the output of the GPC is even or odd. The rank-0 input bit is connected directly to the rank-0 output and is not used within the GPC. There is no need to connect this bit to the GPC input; instead, it should be connected to a GPC at a lower level of the compressor tree.

As an example, consider a (7, 1; 4) GPC. The rank-0 output is always equal to the rank-0 input. The rank-1, -2, and -3 outputs can be computed by a traditional 7:3 counter. Suppose that the rank of this GPC is i. Then, it suffices to eliminate the rank-0 input bit and propagate it to the next level of the tree; then, the GPC is replaced with a 7:3 counter of rank $i + 1$.

m=7, n = 3

Primitive GPCs	Covering GPCs
(0, 0, k; 3) $2 \le k \le 7$	(0, 0, 7; 3)
(0, j, k; 3) $1 \le j \le 3$; $0 \le k \le 7 - 2^j$	(0, 1, 5; 3)
(1, j, k; 3) $0 \le j \le 1$; $0 \le k \le 3 - 2^j$	(0, 2, 3; 3)
	(0, 3, 1; 3)
Reasonable Covering GPCs	(1, 0, 3; 3)
(0, 0, 7; 3) (0, 1, 5; 3) (0, 2, 3; 3)	(1, 1, 1; 3)

Fig. 4. Example of primitive, covering, and reasonable covering GPCs.

Fig. 4 shows the preceding concepts for $m = 7$ and $n = 3$. There are 23 primitive GPCs, 6 covering GPCs, and 3 reasonable covering GPCs. (0, 3, 1; 3) is unreasonable because $K_0 = 1$. (1, 0, 3; 3) is unreasonable because the rank-0 and -1 outputs can be computed by a 3:2 counter, while the rank-2 input connects directly to the rank-2 output; it suffices to replace this GPC with a 3:2 counter of rank i and propagate the rank-2 input bit to the next level of the compressor tree directly. (1, 1, 1; 3) is also unreasonable: Not only is $K_0 = 1$ but also each input bit connects directly to each output; it suffices to propagate these bits directly to the next level of the compressor tree, eliminating this GPC altogether.

D. Configuration Layer

The configuration layer allows the user to program an $m : n$ counter as any m-input n-output reasonable covering GPC. The circuit shown on the right-hand side of Fig. 3 is placed on each configuration layer output. The configuration layer architecture is defined by a set of connections between input ports and muxes. When the right configuration bit is zero, the corresponding counter input is driven to zero; otherwise, it selects one of the inputs connected to the mux.

We make the assumption that m is maximal for a given value of n that satisfies (2), i.e., $m = 2^n - 1$. For example, if $n = 3$, there are 4:3, 5:3, 6:3, and 7:3 counters; based on this assumption, we default to the 7:3 counter.

Let $I = \{I_0, I_1, \ldots, I_{m-1}\}$ and $M = \{M_0, M_1, \ldots, M_{m-1}\}$ be the set of input ports and muxes. A *sensible* configuration layer architecture satisfies the property that each input port I_i is connected to 2^r muxes where r is a nonnegative integer; thus, I_i can be connected to any input bit of rank at most r. r is called the $rank$ of the input port and is denoted $rank(I_i)$. When configuring a GPC, the rank of each input bit connected to input port I_i cannot exceed $rank(I_i)$.

Fig. 5(a) shows an example of a configuration layer (only muxes are shown) for a GPC built from a 15:4 counter. Input ports $I_7, \ldots, I_{14}$ have rank-0; input ports I_0, I_1, I_2, and I_3 have rank-1, and ports I_4, I_5, and I_6 have rank-2.

The configuration layer can be represented as a *configuration graph*, a directed bipartite graph $G = (V, E)$, where $V = I \cup M$ and $E \subseteq I \times M$ represents the set of connections from input ports to muxes, i.e., there is an edge $(I_i, M_j) \in E$ if and only if there is a connection from I_i to M_j. Fig. 5(b) shows an example corresponding to the configuration layer in Fig. 5(a). In Fig. 5(a), I_{13} and I_{14} connect directly to the counter inputs; *dummy muxes* M_{13} and M_{14} are shown in Fig. 5(b) simply to represent the possible connection; a one-input mux in the configuration graph becomes a direct connection in the configuration layer.

E. Configuring the GPC

The configuration graph represents the set of different input-to-mux connections. A *configuration* determines which input port is selected by each mux. At most, one input port can connect to each mux; if no input port is connected, the circuit shown on the right-hand side of Fig. 3 drives the counter input to zero instead. Specifically, a configuration is a subset of edges $C_k \subseteq E$ such that each mux M_j is incident on at most one edge in C_k and each input port I_i. An example of a configuration is the set $\{(I_j, M_j) | 0 \le j \le 14\}$, which configures the GPC as a 15:4 counter. A set of edges including (I_4, M_4) and (I_5, M_4) is not a configuration because two input ports are connected to M_4.

An *active* input port is incident on at least one edge in a configuration. A configuration is *sensible* if each active input port I_i is incident on 2^r edges in C_k, where $0 \le r \le rank(I_i)$. A configuration including edges (I_4, M_4), (I_4, M_5), (I_4, M_6), and (I_7, M_7) is *not* sensible, because the number of ports to which I_4 is connected is not an even power of two.

Let C_k be a configuration, and let $J_r(C_k)$ be the set of input ports that are configured to connect to 2^r counter inputs. To stay within bandwidth limits, the sum of the ranks of the input ports, after configuration, cannot exceed m, the number of counter inputs; in other words

$$\sum_r |J_r(C_k)| \, 2^r \le m. \tag{5}$$

F. Configuration Layer Design

Recall from Section III-A that a GPC is represented as a tuple $P = (K_{N-2}, K_{N-1}, \ldots, K_0; n)$ where K_r is the number of input bits of rank r to be summed, and from Section III-C, recall the definitions of reasonable and covering GPCs.

In this section, we outline a method to design a GPC configuration layer systematically. We do not attempt to achieve universal coverage; instead, we restrict the set of GPCs that can be mapped onto our configurable GPC; doing this allows us to implement the configuration layer with one level of muxes, each having at most two inputs, thereby bounding the delay and area overhead of the configuration layer.

The *rank variation* of a GPC P $rank\_var(P)$ is the number of input bit ranks supported by a GPC, e.g., $N - 1$ for the GPC aforementioned; then, $K_{\mathrm{rank\_var}(P)} \le m/2^r$ in accordance with (5).

A configuration layer can *implement* a GPC P if there is a configuration C_k such that $|J_r(C_k)| \ge K_r$ for $0 \le r \le var(P)$. This is intuitive: P has K_r input bits of rank r, each of which must connect to 2^r muxes. The condition $|J_r(C_k)| \ge K_r$ ensures that a sufficient supply of input ports with the desired connectivity exists.

Let $T_{m,n}$ be the set of reasonable covering GPCs that satisfy I/O constraints $m : n$. A *complete* configuration layer can implement every GPC in $T_{m,n}$. To simplify the design of the configuration layer, we have chosen to restrict the set of reasonable

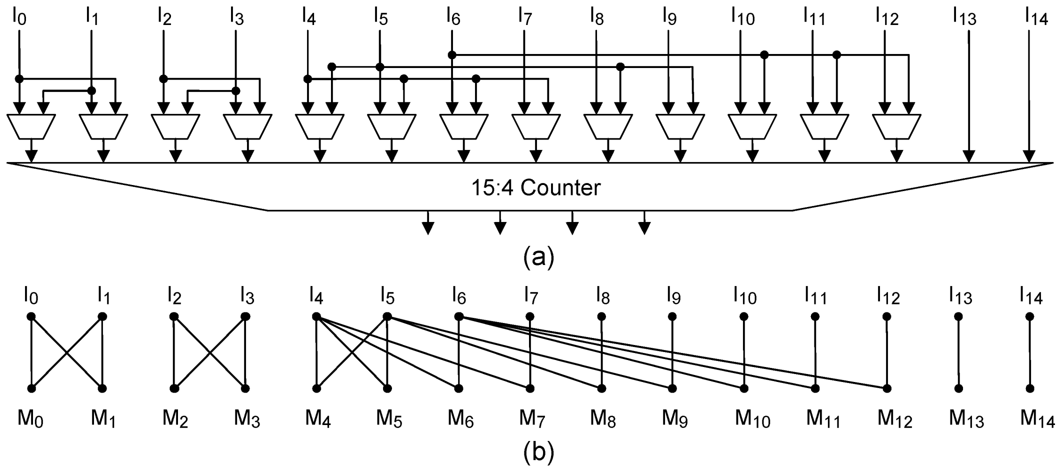

I0 I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11 I12 I13 I14
15:4 Counter
(a)
M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14
(b)

Fig. 5. (a) Configuration layer (muxes only) for a 15-input 4-output GPC. (b) Bipartite graph representation of the configuration layer in (a).

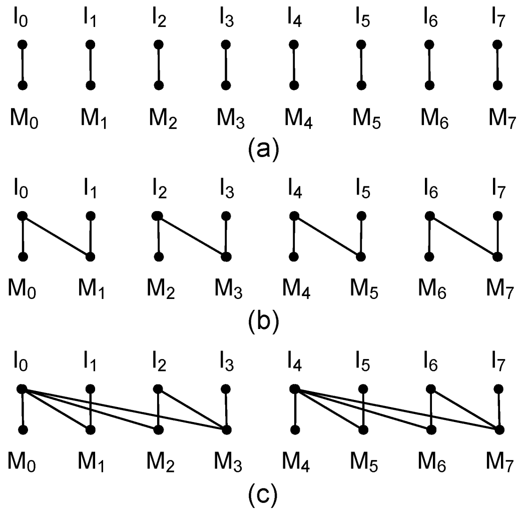


Fig. 6. Configuration graph for (a) S_0, (b) S_0 and S_1, and (c) S_0, S_1, and S_2. In (b), half of the input ports (I_0, I_2, I_4, I_5) are connected to two muxes. In (c), two of the input ports (I_0, I_4) are connected to four muxes. No mux is connected to more than two input ports.

covering GPCs to those whose rank is at most 2; this configuration layer is incomplete, meaning that universal coverage is not achieved.

Let $S_r = \{P \in T_{m,n} | rank\_var(P) = r\}$ be the set of reasonable covering GPCs whose rank variation is r for a given $m : n$. The configuration layer described here must implement S_0, S_1, and S_2. Fig. 6 shows the construction method for an eight-input counter through the incremental addition of edges to a configuration layer graph.

S_0 contains one GPC: All input bits have rank-0, i.e., an $m : n$ counter. Any mapping from input ports to muxes suffices, e.g., $E_0 = \{(I_i, M_i) | 0 \le i \le m-1\}$. Fig. 6(a) shows the initial set of edges.

Now, let us consider S_1. No rank-1 input ports can connect to the same mux; otherwise, both of these input ports could not be configured as rank-1 at the same time. In Fig. 6(b), edges are added to the configuration graph so that input ports I_0, I_2, I_4, and I_6 can be configured as either rank-0 or rank-1.

S_2 contains GPCs having bits of rank-0, -1, or -2. Like the aforementioned reasoning, each rank-2 input port connects to four muxes; $m/2$ of the input ports can already be configured as rank-1; thus, it suffices to take half of them and connect them to two additional muxes. In Fig. 6(c), input ports I_0 and I_4 are extended so that they can be configured as rank-0, -1, or -2. At this point, we stop. In general, there are m input ports in total; $m/4$ connect to four GPC inputs, $m/4$ connect to two, and $m/2$ connect to one.

The basic pattern shown in Fig. 6 is systematic and generalizes to any $m : n$ counter. Stopping at S_2 ensures that the largest mux in the configuration layer has at most two inputs.

IV. FPCA Architecture

A. FPCA Architecture

The FPCA architecture presented by Brisk *et al.* [12] is a 2-D lattice of hierarchical $m : n$ counters connected through a programmable routing network; it had the same basic structure as an island-style FPGA, but with programmable logic cells replaced by $m : n$ counters. The architecture presented here is similar, but programmable GPCs replace the $m : n$ counters. The *connection boxes* that interface each programmable GPC to the adjacent routing channels and *switch boxes* that connect intersecting horizontal and vertical routing channels are the same as an FPGA.

The hierarchical design of an $m : n$ counter increases flexibility. For example, suppose that the counter size in an FPCA is 20:5. This 20:5 counter is hierarchically built from smaller counters, e.g., 4:3. If there are only four input bits to sum at a given time, the smaller counter can be used. This reduces the delay of the circuit at stages of a compressor tree where there is a small number of bits to sum; on the other hand, the large number of smaller counters increases the number of output ports, as several 4:3 counters, for example, will be available. The use of a configurable GPC in lieu of a hierarchically designed $m : n$ counter offers similar flexibility, but without increasing the number of output ports. When there is a small number of bits available at each rank, a GPC can sum bits having different ranks.

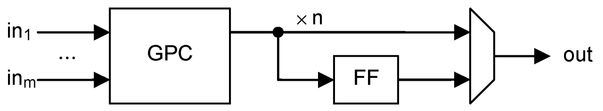


Fig. 7. Flip-flop and mux are placed on each GPC output to allow pipelining.

The programmable GPC described in the preceding section is purely combinational. The user may wish to pipeline the compressor tree in order to increase the clock frequency and throughput. To facilitate this, a flip-flop and mux are placed on each GPC output, as shown in Fig. 7. The same circuit is typically placed on the outputs of FPGA logic blocks (but not on the carry-chain outputs).

In a sense, our intention to embed an FPCA into a FPGA is similar in principle to a cluster of logic cells in a traditional FPGA (e.g., a LAB in an *Altera Stratix*-series FPGA). A LAB (or group of adjacent LABs) could be replaced with an FPCA. The role of a programmable GPC within an FPCA is analogous to the role of a programmable logic cell (an ALM in an *Altera Stratix*-series FPGA) within a LAB. The primary difference is that due to the interconnect structure of compressor trees, a more flexible routing network, in the style of a global (inter-LAB), rather than a local (intra-LAB), FPGA routing network is required for the FPCA.

B. FPCA Mapping Heuristic

The algorithm to map a compressor tree onto an FPCA is based on a heuristic developed by Parandeh-Afshar *et al.* [9] to map compressor trees onto the logic cells (six LUTs) of high-performance FPGAs. The FPCA is homogeneous, i.e., $m : n$ is the same for all programmable GPCs. For a given $m : n$, the available GPCs are the set of reasonable covering GPCs whose rank variation does not exceed 2. No further modifications to the mapping heuristic are required.

C. FPCA Alternatives

Here, we qualitatively explore several different alternative methods to integrate $m : n$ counters into an FPGA and explain why we believe that the FPCA is superior.

One possibility would be to add a programmable GPC to a LAB or to replace one of the ALMs with the programmable GPC; however, one GPC is unlikely to be used in isolation: A collection of them is required to construct a compressor tree. Thus, a compressor tree synthesized on this architecture would not be able to take advantage of the fast local connections within the LAB. A better approach is to cluster the GPCs together, as is done by the FPCA. Replacing all of the ALMs in a LAB with programmable GPCs effectively yields an FPCA with a local LAB style rather than a global routing network; we have opted for a routing network in the global style due to the complex interconnect structure required to construct a compressor tree from GPCs.

A second alternative is to integrate a GPC into an ALM as a programmable type of macroblock, similar in principle to the work by Cong and Huang [36] and Hu *et al.* [37]; however, this architecture significantly increases the input and output bandwidth of the ALM; it is unlikely that the local routing network within a LAB could handle this increased I/O bandwidth as it exists today. We believe that a better approach is to strictly separate the FPCA/GPCs from the LAB/ALMs.

V. Experimental Setup

A. VPR

Two different versions of the Versatile Place-and-Route (VPR) tool [38], [39] were used to evaluate the FPCA. The most recent version of VPR, version 5.0, was used to compare the performance advantages of an FPGA containing an FPCA against an FPGA containing DSP blocks as a baseline. The earlier version of VPR does not support DSP blocks or any type of embedded IP core; therefore, the newer version was required to perform this comparison.

An earlier version, version 4.30, was used for a comparison of energy consumption. At present, no power model is currently available for the newer version of VPR; as discussed in Section V-C, we extended a preexisting power model for the earlier version to compute the energy consumption.

VPR 5.0 provides preconstructed architecture models for different process technologies; VPR 4.30, in contrast, requires the user to provide transistor-level properties of the wires. Details will be provided in the following section.

B. Delay and Area Extraction

The FPCA was modeled as a stand-alone device using VPR 4.30. Each compressor tree in each benchmark was extracted and synthesized on the FPCA. The FPCA was then modeled as an IP core in VPR 5.0; for each benchmark, the delay through each path through the FPCA was taken from VPR 4.30. The complete benchmark was then synthesized on VPR 5.0, with all compressor trees mapped onto FPCAs. The total delay includes both non-compressor tree logic mapped onto the general logic of the FPGA along with the compressor tree delay through the FPCA. To model the FPCA, the traditional FPGA logic blocks in VPR 4.30 were replaced with programmable GPCs. After mapping a compressor tree onto a network of GPCs, VPR was used to place-and-route the circuit. VPR also reported the critical path delay, which includes both routing and logic delays. The number of GPCs required to synthesize each compressor tree can be determined from the result of the mapping heuristic.

The programmable GPCs described in Section III were modeled in Very High Speed Integrated Circuit Hardware Description Language (VHDL) and synthesized using *Synopsys Design Compiler* with 90-nm *TSMC* standard cells. *Cadence Silicon Encounter* was then used to place and route the designs and extract delay and area estimates. This was done for four different programmable GPCs, with $m : n = 8 : 4$, 12:4, 16:5, and 20:5. Thus, four different FPCA architectures were studied, as the GPC size is assumed to be homogeneous within an FPCA. A separate VPR architecture description file (ADF) was instantiated for each FPCA. We limited the channel width to 40 segments, VPR's default value.

For the purpose of comparison, we modeled an island-style FPGA whose logic blocks resemble Altera's ALM and whose logic clusters resemble Altera's LAB, but with four blocks per cluster; the limited number of ALMs per LAB was due to complications involved in modeling carry chains in VPR. Each LAB has two carry chains to support ternary addition. The primary difference between this baseline architecture and the Stratix II and III is that the baseline is island style, while the Stratix II and III organize LABs into columns and employ nonuniform routing in the x- and y-directions. The GPC mapping heuristic of Parandeh-Afshar *et al.* [9] was used to synthesize compressor trees onto this FPGA.

To model routing delays, VPR 4.30 requires information such as the per-unit resistance and capacitance of wires. Our experiments used *TSMC* 90-nm technology, and the per-unit resistance and capacitance for metal-6 were computed and inserted into VPR's ADF. These values were used to compute the routing delays of the FPCA. The per-unit resistance and capacitance of metal-6 were chosen, as this metal layer seemed to be a reasonable choice for the wires in the routing network; in practice, the routing network is likely to be realized in several metal layers.

VPR 5.0, in contrast, provides preconstructed architecture models for different transistor technologies. We used an appropriate model, which eliminated the need to explicitly provide per-unit resistance and capacitances of the wires.

C. VPR Power Model

A power model for VPR was developed by Poon *et al.* [40] to model traditional island-style FPGAs. Choy and Wilton [41] modified this framework to support power estimation for embedded IP cores, such as DSP blocks. We extended these models to estimate the power consumption of the FPCA. The Activity Estimator [42] estimates the probability of transitions occurring in the circuits mapped to an FPGA; static probability and density probability [43] are used to extract the transition activities of each net. The complexity of this computation is $O(2^k)$, where k is the number of inputs. This time complexity is suitable for LUT-based logic blocks, where the number of inputs is typically six or less (ignoring carry chains). The programmable GPCs used in our study of FPCAs have up to 20 inputs; for circuits of this size, the exponential runtime of the model becomes a limiting factor.

D. FPCA Power Model for VPR

Due to the complexity of the VPR power model described in the preceding section, we developed a more efficient simulation-based power model. Our power model is based on the *Lookup* technique advocated by Choy and Wilton [41].

The power model consists of an offline power characterization of each GPC under different input switching activity probabilities. The results are collected in a table and fed into an online power estimator that extracts the switching activities via simulation. The simulator dynamically accesses the LUT to determine the power dissipated given the switching activity at each point in the simulation.

The offline power characterization flow is described as follows. First, the programmable GPC is modeled in VHDL and synthesized using *Synopsys Design Compiler*. Object and node names are extracted; these names are later used in the simulation phase for assigning switching activities. The power of the programmable GPC is estimated using *Synopsys PrimePower*; the power characteristics of the GPCs are extracted with different transition activity rates. These rates are then organized into tables, indexed by the transition probabilities, which are then input into the online flow.

The online power estimator begins with a mapped netlist whose objects and nodes are extracted. The objects are the GPC blocks used for mapping a compressor tree. The transition activities of objects are extracted through the application of stimulus vectors, which are generated randomly. As the accuracy of the power calculations used by VPR depends on the accuracy of the switching activity annotated to the design, it is essential to achieve high covering during simulation. High coverage is achieved via simulation feedback to the random vector generator. After a set of random vectors with high signal coverage is found, the simulator computes the activity transitions of objects and nets listed in the object list.

Next, a modified version of VPR is used to estimate the power dissipated by the FPCA. VPR's power model is based on transition density and the static probability of nets. The transition density of a signal represents the average number of transitions of that signal per unit time; the static probability of a net is the probability that the signal is high at any given time. These two parameters are computed for each net in the design using the simulation output.

The power model is used to extract the power model for the FPCA. The offline GPC power characteristics are placed as a table in the ADF that describes the FPCA. The table contains the estimated power dissipation for transition activities ranging from 0 to 1 by increments of 0.1; separate tables are instantiated depending on whether or not each output of the GPC is written to its flip-flop.

A second input to the power model is the transition activity of objects extracted in the previous step. VPR reports three different power estimates: the dynamic power dissipated by the GPC and by the routing network and the leakage power. The power consumption of the routing network is estimated using switching activities and switch box and wire parameters specified in the ADF based on the target technology. The GPC power consumption is estimated using the average activity of its inputs and the offline power table in the ADF.

VI. Experimental Results

A. Benchmarks

We selected a set of benchmarks from arithmetic, DSP, and video processing domains where we were able to identify compressor trees. These benchmarks were broadly categorized into *multiplier-based* and *multi-input addition* benchmarks.

The multiplier-based benchmarks include *g.721* [44], a polynomial function that has been optimized using *Horner's Rule (hpoly)*, 10×10- and 20×20-b multipliers ($m10 \times 10$ and $m20 \times 20$), and a video processing application (*video mixer* [45]). The *video mixer* converts two channels of red–green–blue video to television-standard YIQ signals and then mixes them in an alpha blender to produce a composite output signal.

The multi-input addition benchmarks include the *MediaBench* application *adpcm* [44], a 1-D multiplierless discrete cosine transform (*dct* [8]), three- and six-tap finite-impulse

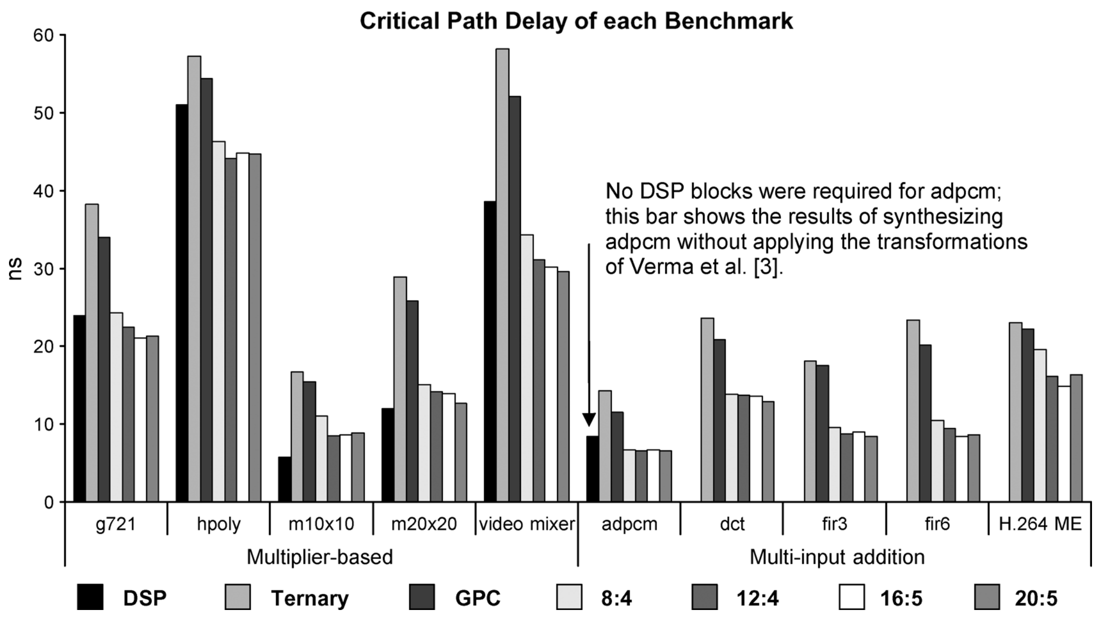


Fig. 8. Critical path delay observed for each benchmark without the transformations of Verma and Ienne [11] and with multiplication operations synthesized on DSP blocks (DSP); all other synthesis methods applied the transformations. Ternary and GPC synthesize each benchmark wholly on the general logic of the FPGA, while 8:4, 12:4, 16:5, and 20:5 synthesize each compressor tree on an FPCA that is integrated into a larger FPGA.

response filters with randomly generated constants (*fir3* and *fir6* [6]), and an internally developed variable block size motion estimator for H.264/AVC video coding (*H.264 ME*).

The multiplier-based benchmarks contain multipliers that can be synthesized on an FPGA using DSP blocks; however, when the transformations of Verma *et al.* [3] are applied, the compressor trees within the multipliers are merged with other addition operations, rendering the DSP blocks useless. After applying these transformations, the compressor trees within these benchmarks can only be synthesized on the general logic of the FPGA or on an FPCA.

The *video mixer* contains many disparate compressor trees, even after the transformations of Verma *et al.* are applied; all other benchmarks contain one compressor tree. *H.264 ME* contains a set of identical processing elements (PEs), where each PE contains a compressor tree. The number of PEs can vary depending on the needs of the system. We chose to synthesize a four-PE system, ignoring the memory and control logic.

Each benchmark was synthesized six or seven times.

1) **DSP**: The multiplier-based benchmarks and *adpcm* were synthesized without applying the transformations of Verma *et al.* All multipliers in the multiplier-based benchmarks were synthesized on the DSP blocks. *adpcm* contains three disparate addition operations, but cannot use DSP blocks. In all subsequent experiments, the transformations of Verma *et al.* were applied to the multiplier-based benchmarks and to *adpcm*; the remaining multi-input addition benchmarks were written with compressor trees explicitly exposed. DSP blocks cannot be used for multiplication operations following the transformations.
2) **Ternary**: Compressor trees are synthesized on ternary adder trees using FPGA logic cells configured as ternary adders.
3) **GPC**: Compressor trees are synthesized on the general logic of an FPGA using the GPC mapping heuristic of Parandeh-Afshar *et al.* [9].
4) **8:4, 12:4, 16:5, and 20:5**: The compressor tree is synthesized on an FPCA; four different FPCAs with different counter sizes were considered.

The experiments synthesized purely combinational circuits. In actuality, the frequency and throughput of a compressor tree could be increased by registering the output bits of each level of logic in the tree. The benchmarks that were implemented did not naturally contain pipelined compressor trees; therefore, this possibility is not evaluated here.

Lastly, we note that Brisk *et al.* [12] attempted to synthesize adder trees using the DSP blocks; this approach yielded very slow compressor trees; as such, these experiments are not repeated here, as the approach is not competitive.

B. Results

Fig. 8 shows the critical path delay of each benchmark after synthesis. In all cases, other than the two multipliers ($m10 \times 10$ and $m20 \times 20$), the FPCA yields the minimum critical path delay. In particular, the FPCA's success on the multiplier-based benchmarks compared with that of a DSP is due to its ability to accelerate compressor trees generated by the transformations of Verma *et al.* $m10 \times 10$ and $m20 \times 20$ do not benefit from these transformations, as their compressor trees are not merged with any other operations.

It should be noted that $m10 \times 10$ and $m20 \times 20$ are worst case examples for the *Altera*-style FPGA. The reason is that each half-DSP block contains a 9×9-b multiplier; for example, four 9×9-b multipliers are required for $m10 \times 10$. The gap between the DSP block and the FPCA would be exacerbated for 9×9- and 18×18-b multiplication.

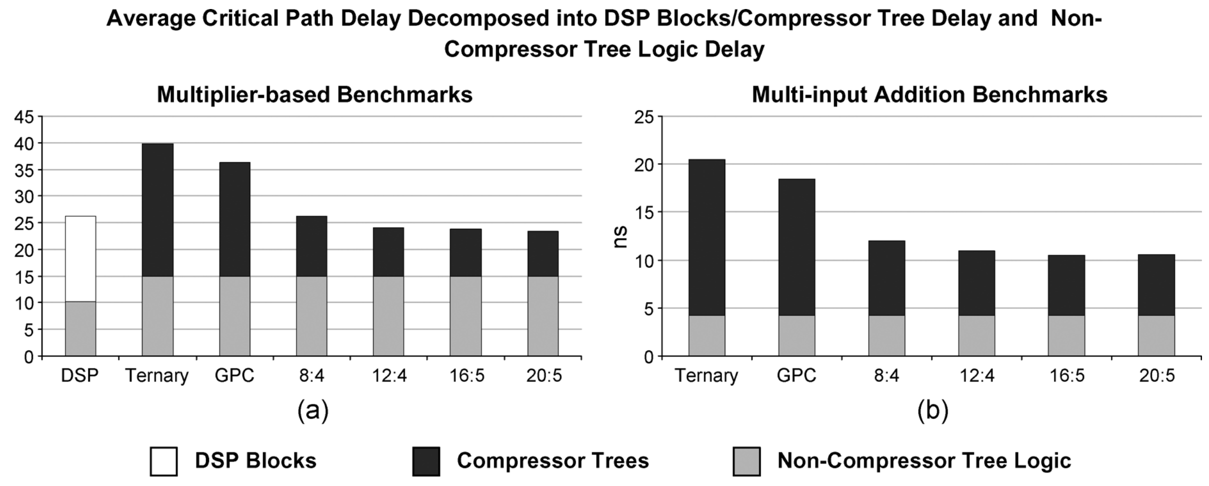


Fig. 9. Average delay of the set of benchmarks decomposed into the delay through DSP blocks (DSP only)/compressor tree logic (all others) and non-compressor tree logic (all synthesis methods) for the (a) multiplier-based and (b) multi-input addition benchmarks.

The benefits of the FPCA are exacerbated for the *video mixer* because Verma *et al.*'s transformations are particularly effective for this benchmark: it has many multiplication and addition operations that are merged together by these transformations.

Fig. 8 also shows that the FPCA considerably reduces the critical path delay over *Ternary* and GPC; this is particularly important for the multi-input addition benchmarks, where the DSP blocks cannot be used.

Among the different FPCA options, none performed uniformly better than the others. In the case of *adpcm*, all four FPCAs achieved comparable critical path delays. Among the FPCAs, 12:4 had the minimum critical path delay for *hpoly* and $m10 \times 10$; 16:5 had the minimum critical path delay for *g721*, *fir6*, and *H.264 ME*; and 20:5 had the minimum critical path delay for the remaining benchmarks. These results indicate that no FPCA will be ideal for all benchmarks, but the counter size should probably be larger than 8:4.

Fig. 9 shows the delay achieved by synthesizing each benchmark into DSP block/compressor tree logic and non-compressor tree logic. Fig. 9(a) shows the multiplier-based benchmarks, where DSP blocks can be used, and Fig. 9(b) shows the results for the multi-input addition benchmarks. Due to its limited functionality, the FPCA only speed up the delay of the compressor tree.

In Fig. 9(a), the FPCA reduces the average compressor tree logic delay from 30% (8:4) to 47% (20:5); however, the transformations of Verma *et al.* and the need to synthesize partial-product generators on the FPGA general logic when DSP blocks are not used increase the average non-compressor tree logic delay by 46%. GPC and *Ternary* increase the average critical path delay compared to DSP; 8:4, 12:4, 16:5, and 20:5 reduce the average critical path delay compared with DSP by 0.2%, 8%, 10%, and 11%, respectively.

In Fig. 9(a), the increase in average non-compressor tree logic delay for all options other than DSP is due to the fact that partial-product generators must be synthesized on general FPGA logic, rather than the DSP blocks. On the other hand, the FPCA noticeably reduces the average delay of the resulting compressor trees considerably compared with *Ternary* and GPC.

DSP blocks cannot be used for the multi-input addition benchmarks in Fig. 9(b); we take GPC as a baseline as its critical path delay is less than ternary. Since Verma and Ienne's transformations are applied to *adpcm* and the other multi-input addition benchmarks have compressor trees directly exposed, the non-compressor tree logic delay is the same in all cases. Compared with GPC, 8:4, 12:4, 16:5, and 20:5 reduce the overall (compressor tree) delay by 35% (45%), 41% (53%), 43% (56%), and 43% (55%), respectively. As there are no partial-product generators and DSP blocks are not used, these benefits are due solely to critical path reduction within the compressor trees.

Fig. 10 shows the area of each benchmark converted to two-input NAND gate equivalents (GEs); the area includes the computational elements (LUTs, DSP blocks, and GPCs) and does not include any estimates of the utilization of resources in the programmable routing network.

Each DSP block contains eight 9×9-b multipliers and has an area of 10 714 GEs. An $m \times n$-bit multiplier generates mn partial products. Since each ALM produces two output bits, $mn/2$ ALMs are required. In theory, this gives DSP blocks an advantage in terms of area utilization compared with the other methods.

For *hpoly*, $m10 \times 10$, and $m20 \times 20$, the FPCA consumed considerably more area than DSP. This is due, primarily, to the fact that partial-product generators must be synthesized on the general logic of the FPGA. It should be noted that $m10 \times 10$ required just one DSP block but only used four of the eight multipliers. In other cases, namely, *g721* and the *video mixer*, the FPCAs had similar area requirements to DSP; however, 12:4 for video mixer was significantly smaller. For this benchmark, GPCs built from 12:4 counters, coincidentally, were the perfect-sized building blocks.

Compared to GPC and *Ternary*, the FPCAs reduced the area requirement; in most cases, the area reduction was marginal; however, it was quite pronounced for the *video mixer* and *fir6*. Similar to the critical path delay results reported in Fig. 8, none of the four FPCA options was uniformly better than the others across all benchmarks.

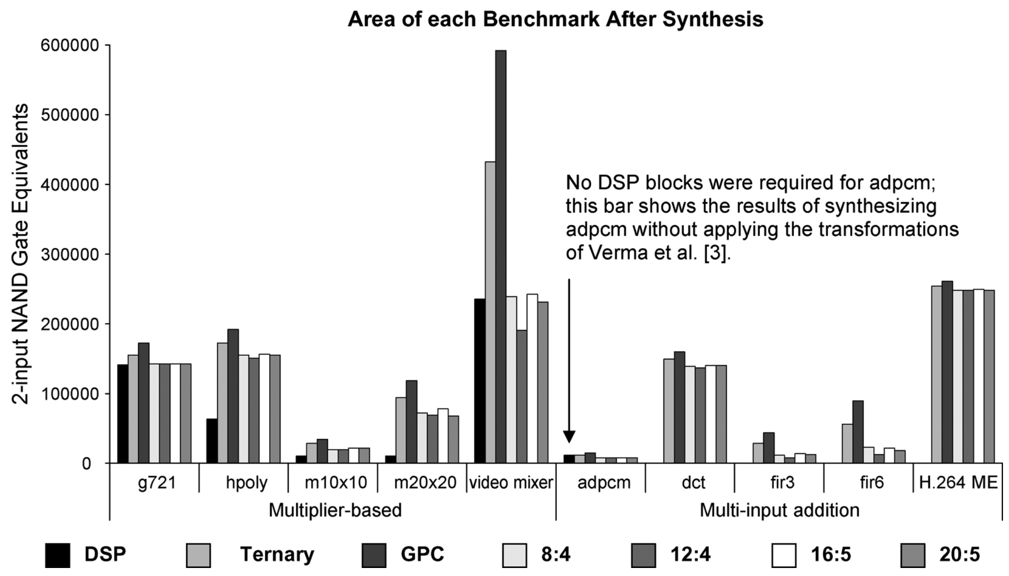


Fig. 10. Area of each benchmark after synthesis. The area of the DSP blocks, LUTs, and GPCs have been converted to two-input NAND GEs. These area estimates do not account for the programmable routing network.

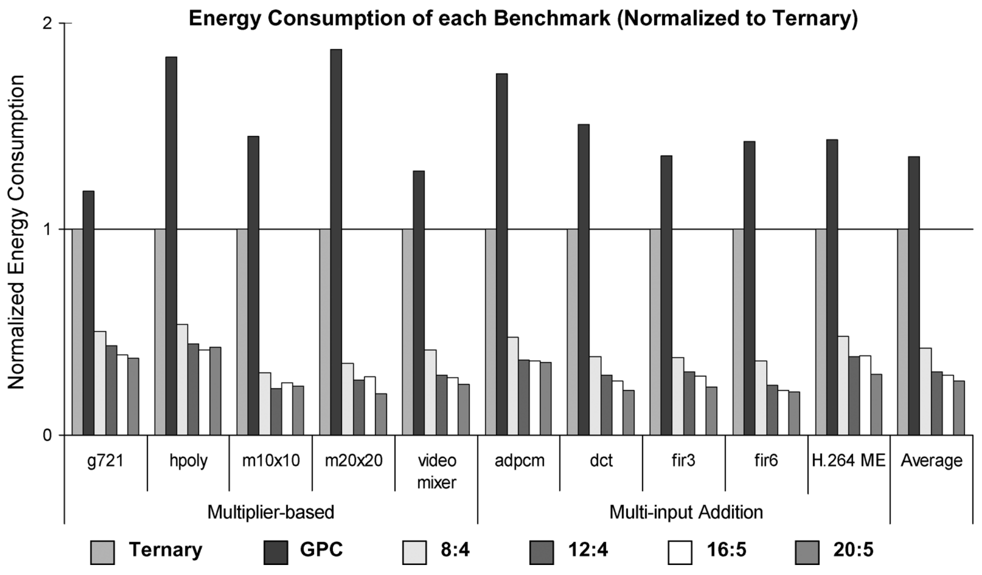


Fig. 11. Energy consumption of each benchmark (normalized to Ternary).

Fig. 11 shows the normalized energy consumption of each benchmark. As VPR 5.0 does not have a power model, the energy consumption reported here are only for the compressor trees and were measured using VPR 4.30. No energy consumption for DSP is reported since VPR 4.30 does not support embedded blocks.

Fig. 12 shows the average energy consumption across the set of benchmarks, decomposed into energy consumed by the logic elements (ALMs/GPCs) and the routing network.

GPC consumes more energy than the other options. GPC builds a compressor tree using six-input GPCs with three or four outputs; two ALMs per GPC are required. Each ALM in *Ternary*, in contrast, takes six input bits and produces two output bits (ignoring the carry-out bit, which is propagated to the next ALM in the chain). For this reason, GPC tends to require more ALMs and dissipates more static power.

Fig. 12 shows that the primary advantage of the FPCA comes from its ability to reduce logic delay. In both *Ternary* and GPC, LUT-based ALMs are used to realize the arithmetic building blocks for compressor trees. The FPCA, in contrast, uses ASIC implementations of these components, which is considerably more efficient. Although *Ternary* consumes less energy in the routing network than any of the alternatives, the FPCA more than makes up for this in terms of energy savings in the logic. In conclusion, both Figs. 11 and 12 show that the FPCA significantly reduces energy consumption compared to *Ternary* and GPC.

We suspect that DSP blocks will consume less energy for multiplication operations, because the other methods will need to synthesize the partial-product generators on the general logic for the FPGA and the number of partial products per multiplication operation is quadratic.

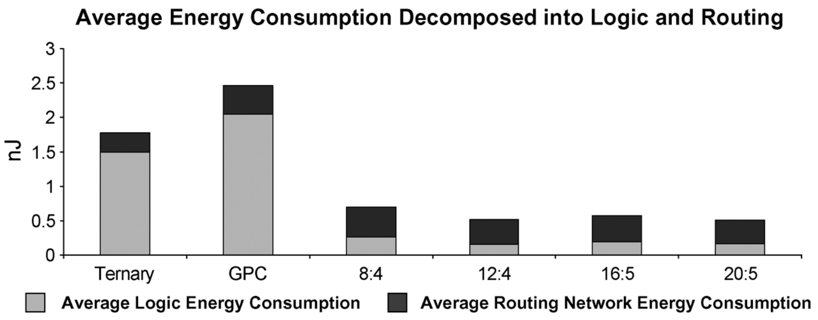


Fig. 12. Average energy consumption due to logic and routing.

VII. Conclusion and Future Work

The FPCA is a programmable IP core that can accelerate compressor trees on FPGAs. For parallel multiplication, the FPCA retains many of the advantages of the embedded multipliers in the DSP blocks; however, it suffers a disadvantage in terms of area utilization because the partial products must be synthesized on the FPGA general logic. For parallel multiplication, the FPCA retains most of the benefits of the embedded multipliers in the DSP blocks, while providing a variable-bitwidth solution for multiplication operations that do not match the fixed bitwidth of the DSP blocks. Moreover, the FPCA can accelerate multi-input addition operations, while the DSP blocks cannot, particularly when used in conjunction with transformations by Verma *et al.* [3] to expose compressor trees at the application level. Furthermore, the FPCA reduces the critical path delay and energy consumption compared to the best methods to synthesize compressor trees on the FPCA.

The DSP block will generally outperform the FPCA for applications containing many multiplications whose bitwidths match precisely that of the ASIC multipliers in the embedded DSP blocks and for which the transformations of Verma *et al.* are ineffective. For virtually all other applications that contain compressor trees—naturally or via transformation, the FPCA performs significantly better than current FPGAs.

References

[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 6, p. 754, Dec. 1964.

[2] L. Dadda, "Some schemes for parallel multipliers," *Alta Freq.*, vol. 34, pp. 349–356, Mar. 1965.

[3] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1761–1774, Oct. 2008.

[4] S. Mirzaei, A. Hosangadi, and R. Kastner, "FPGA implementation of high speed FIR filters using add and shift method," in *Proc. Int. Conf. Comput. Des.*, San Jose, CA, Oct. 2006, pp. 308–313.

[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 Syst. I, Reg. Papers*, vol. 53, no. 2, pp. 578–593, Feb. 2006.

[6] 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," in *Proc. IEEE Custom Integr. Circuits Conf.*, San Jose, CA, Sep. 2005, pp. 59–62.

[7] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar, and A. Alvandpour, "A 6.2-Gflops floating-point multiply-accumulator with conditional normalization," *IEEE J. Solid-State Circuits*, vol. 41, no. 10, pp. 2314–2323, Oct. 2006.

[8] A. Shams, W. Pan, A. Chandanandan, and M. Bayoumi, "A high-performance 1D-DCT architecture," in *Proc. IEEE Int. Symp. Circuits Syst.*, Geneva, Switzerland, May 2000, vol. 5, pp. 521–524.

[9] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient synthesis of compressor trees on FPGAs," in *Proc. Asia-South Pacific Des. Autom. Conf.*, Seoul, Korea, Jan. 2008, pp. 138–143.

[10] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Improving synthesis of compressor trees on FPGAs via integer linear programming," in *Proc. Int. Conf. Des. Autom. Test Eur.*, Munich, Germany, Mar. 2008, pp. 1256–1261.

[11] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[12] P. Brisk, A. K. Verma, P. Ienne, and H. Parandeh-Afshar, "Enhancing FPGA performance for arithmetic circuits," in *Proc. Des. Autom. Conf.*, San Diego, CA, Jun. 2007, pp. 404–409.

[13] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, "A compact high-speed parallel multiplication scheme," *IEEE Trans. Comput.*, vol. C-26, no. 10, pp. 948–957, Oct. 1977.

[14] S. Dormido and M. A. Canto, "Synthesis of generalized parallel counters," *IEEE Trans. Comput.*, vol. C-30, no. 9, pp. 699–703, Sep. 1981.

[15] S. Dormido and M. A. Canto, "An upper bound for the synthesis of generalized parallel counters," *IEEE Trans. Comput.*, vol. C-31, no. 8, pp. 802–805, Aug. 1982.

[16] *"Stratix III Device Handbook, Vol. 1 and 2"* Altera Corporation, San Jose, CA, Feb. 2009. [Online]. Available: http://www.altera.com/

[17] *"Virtex-5 User Guide"* Xilinx Corporation, San Jose, CA, 2007. [Online]. Available: http://www.xilinx.com/

[18] *"Virtex-5 FPGA Xtreme DSP Design Considerations"* Xilinx Corporation, San Jose, CA, Jan. 2009. [Online]. Available: http://www.xilinx.com/

[19] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, "A hybrid ASIC and FPGA architecture," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2002, pp. 187–194.

[20] P. Jamieson and J. Rose, "Architecting hard crossbars on FPGAs and increasing their area-efficiency with shadow clusters," in *Proc. IEEE Int. Conf. Field Programmable Technol.*, Kitakyushu, Japan, Dec. 2007, pp. 57–64.

[21] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Architectural modifications to enhance the floating-point performance of FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 2, pp. 177–187, Feb. 2008.

[22] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid-reconfigurable systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, pp. 602–627, Oct. 2002.

[23] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. K. Verma, P. Brisk, F. Gurkaynak, Y. Leblebici, and P. Ienne, "Architecture improvements for field programmable counter arrays: Enabling synthesis of fast compressor trees on FPGAs," in *Proc. Int. Symp. FPGAs*, Monterey, CA, Feb. 2008, pp. 181–190.

[24] D. Cherepacha and D. Lewis, "DP-FPGA: An FPGA architecture optimized for datapaths," *VLSI Des.*, vol. 4, no. 4, pp. 329–343, 1996.

[25] A. Kaviani, D. Vranseic, and S. Brown, "Computational field programmable architecture," in *Proc. IEEE Custom Integr. Circuits Conf.*, Santa Clara, CA, May 1998, pp. 261–264.

[26] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 2, pp. 138–147, Apr. 2000.

[27] K. Leijten-Nowak and J. L. van Meerbergen, "An FPGA architecture with enhanced datapath functionality," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 2003, pp. 195–204.

[28] M. T. Frederick and A. K. Somani, "Multi-bit carry chains for high-performance reconfigurable fabrics," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Madrid, Spain, Aug. 2006, pp. 1–6.

[29] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "A novel FPGA logic block for improved arithmetic performance," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 2008, pp. 171–180.

[30] B. Parhami, "Configurable arithmetic arrays with data-driven control," in *Proc. Asilomar Conf. Signals, Syst., Comput.*, Pacific Grove, CA, Oct./Nov. 2000, pp. 89–93.

[31] R. Francis, S. Moore, and R. Mullins, "A network of time-division multiplexed wiring for FPGAs," in *Proc. 2nd IEEE Symp. Networks-on-Chip*, Apr. 2008, pp. 35–44, Newcastle University, U.K..

[32] N. L. Miller and S. F. Quigley, "A novel field programmable gate array architecture for high speed arithmetic processing," in *Proc. 8th Int. Workshop Field-Programmable Logic Appl.*, Tallinn, Estonia, Aug./Sep. 1998, pp. 386–390.

[33] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 1999, pp. 135–143.

[34] S. Fiske and W. J. Dally, "The reconfigurable arithmetic processor," in *Proc. 15th Int. Symp. Comput. Archit.*, Honolulu, HI, May/Jun. 1988, pp. 30–36.

[35] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, Sep./Oct. 2007.

[36] J. Cong and H. Huang, "Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 1, pp. 3–23, Jan. 2005.

[37] Y. Hu, S. Das, S. Trimberger, and L. He, "Design, synthesis and evaluation of heterogeneous FPGA with mixed LUTs and macro-gates," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2007, pp. 188–193.

[38] V. Betz and J. Rose, "VPR: A new packing, placement, and routing tool for FPGA research," in *Proc. 7th Int. Workshop Field-Programmable Logic Appl.*, London, U.K., Sep. 1997, pp. 213–222.

[39] V. Betz, J. Rose, and A. Marquardt*, Architecture and CAD for Deep Submicron FPGAs*. Norwell, MA: Kluwer, Feb. 1999.

[40] K. K. W. Poon, S. J. E. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 2, pp. 279–302, Apr. 2005.

[41] N. C. K. Choy and S. J. E. Wilton, "Activity-based power estimation and characterization of DSP and multiplier blocks in FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Technol.*, Bangkok, Thailand, Dec. 2006, pp. 253–256.

[42] J. Lamoureux and S. J. E. Wilton, "Activity estimation for field programmable gate arrays," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl.*, Madrid, Spain, Aug. 2006, pp. 1–8.

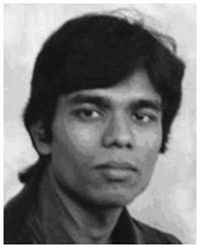
[43] F. N. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 4, pp. 446–455, Dec. 1994.

[44] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Int. Symp. Microarchitecture*, Research Triangle Park, NC, Dec. 1997, pp. 330–335.

[45] *"Creating High-Speed Data Path Components—Application Note,"* Synopsys Corporation, Mountain View, CA, Oct. 1999, ver. 1999.10, Chapter 1.

Hadi Parandeh-Afshar received the B.S. degree in computer engineering and the M.S. degree in computer architecture from the University of Tehran, Tehran, Iran, in 2001 and 2003, respectively. He has been working toward the Ph.D. degree in the Processor Architecture Laboratory, School of Computer and Communications Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, since 2007.

His research interests include reconfigurable computing, computer architecture and arithmetic, and design automation for embedded systems.



Ajay Kumar Verma received the B.S. degree in computer science from the Indian Institute of Technology Kanpur, Kanpur, India, in 2003. He has been working toward the Ph.D. degree in the Processor Architecture Laboratory, School of Computer and Communications Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, since 2004.

His research interests include logic synthesis, optimization of arithmetic circuits, and design automation for application-specific processors.

Mr. Verma was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems in 2007.

Philip Brisk received the B.S., M.S., and Ph.D. degrees from the University of California, Los Angeles, in 2002, 2003, and 2006, respectively, all in computer science.

Since 2006, he has been a Postdoctoral Scholar with the Processor Architecture Laboratory, School of Computer and Communications Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. His research interests include reconfigurable computing, compilers, and design automation and architecture for application-specific processors.

Dr. Brisk is or has been a member of the program committees of several international conferences and workshops, including the IEEE Symposium on Application-Specific Processors, the International Workshop on Software and Compilers for Embedded Systems, and the Reconfigurable Architecture Workshop. He was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems in 2007.

Paolo Ienne (S'94–M'96) received the Dottore degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1991 and the Ph.D. degree from the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996.

In December 1996, he joined the Semiconductors Group, Siemens AG, Munich, Germany (which later became Infineon Technologies AG), where after working on data path generation tools, he became the Head of the Embedded Memory Unit, Design Libraries Division. Since 2000, he has been with EPFL, where he is currently a Professor and heads the Processor Architecture Laboratory, School of Computer and Communications Sciences. His research interests include various aspects of computer and processor architecture, computer arithmetic, reconfigurable computing, and multiprocessor systems-on-chip.

Dr. Ienne is or has been a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the International Conference on Computer Aided Design, the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), the International Symposium on Low Power Electronics and Design, the International Symposium on High-Performance Computer Architecture, the International Conference on Field Programmable Logic and Applications, and the IEEE International Symposium on Asynchronous Circuits and Systems. He was the General Cochair of the Sixth IEEE Symposium on Application-Specific Processors (SASP'08) and a Guest Editor for a special section of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS on Application Specific Processors. He was a recipient of the Design Automation Conference 2003 and the CASES 2007 Best Paper Awards.