Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits

Ajay K. Verma, Philip Brisk, and Paolo Ienne, *Member, IEEE*

***Abstract*—The increasing importance of datapath circuits in complex systems-on-chip calls for special arithmetic optimizations. The goal is to automatically achieve the handcrafted results which escape classic logic optimizations. Some work has been done in the recent years to infer the use of the carry-save representation in the synthesis of arithmetic circuits. Yet, many cases of practical interest cannot be handled due to the scattering of logic operations among the arithmetic ones—particularly in arithmetic computations which are originally described at the bit level in high-level languages such as C. We therefore introduce an algorithm to restructure dataflow graphs so that they can be synthesized as high-quality arithmetic circuits, close to those that an expert designer would conceive. On typical embedded software benchmarks which could be advantageously implemented with hardware accelerators, our technique always reduces tangibly the critical path by up to 46% and generally achieves the quality of manual implementations. In many cases, our algorithm also manages to reduce the cell area by up to 10%–20%.**

***Index Terms*—Arithmetic optimization, logic synthesis, term rewriting system.**

I. Introduction and Goal

THE PROLIFERATION of complex digital signal processing in most electronic consumer products and the need of implementing these demanding applications at minimal cost in area and energy make efficient synthesis of datapaths particularly important. Unfortunately, good quality implementations are particularly difficult to achieve automatically, especially when the original specification of the functionality required is, as it is often the case, in the form of a software implementation. Computations described at bit level through typical programming constructs (e.g., software implementations of limited precision fixed point, *ad-hoc* limited-precision floating point, and bit- or byte-level manipulations typical of cryptography) are particularly resistant to known optimization techniques.

Fig. 1 shows a typical example of computational kernel; the figure represents the dataflow graph of the original description in C language of the kernel of `adpcmdecode` [1]. The dataflow represents a slightly approximated 16×4 multiplier, but its direct implementation, even applying known techniques described in literature, would produce a suboptimal result due to a sequence of carry-propagate additions interspersed with multiplexers (SEL nodes in the figure). An expert designer would instead replace the additions with a Wallace-like compressor tree followed by a single carry-propagate final adder [2]. Our goal is to achieve the speed of the latter implementation with automatic transformations of the dataflow graph. The result of applying these transformations on this example is shown in Fig. 27 later in this paper.

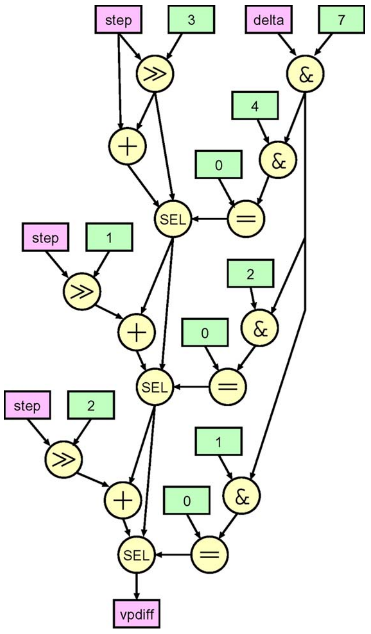


Fig. 1. Typical dataflow graph obtained by parsing automatically the C-language bit-level description of a custom arithmetic operator.

In the next section, we review previous work relevant to our goal. In the following section, we describe our optimization algorithm. Then, we compare our results on several benchmarks with plain synthesis and with a commercial arithmetic optimizer. We discuss future work in the concluding section.

II. Related Work

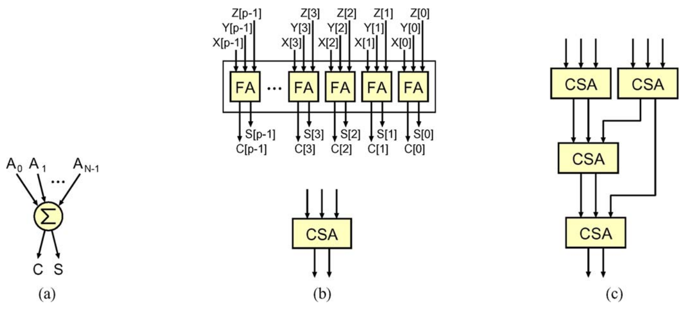
The use of carry-save representation to build fast multiple-input adders is a traditional arithmetic design technique, usually applied to parallel multipliers [2]. A large body of literature exists on the best way to build the compressor tree of parallel multipliers, and its analysis goes beyond the scope of this review. Among the major contributions, the work of Stelling *et al.* [3] discusses various heuristic and optimal algorithms to design arbitrary compressor trees (see Fig. 2). This paper focuses on exposing compressor trees in arithmetic computations, and we use a simple heuristic similar to the three-greedy approach [3], [4] to demonstrate the strength of our

Manuscript received March 6, 2008. Current version published September 19, 2008. This paper was recommended by Associate Editor R. Camposano.

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

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2003280



A0 A1 ... AN-1
Σ
C S
Z[p-1] Y[p-1] X[p-1]
Z[3] Y[3] X[3] Z[2] Y[2] X[2] Z[1] Y[1] X[1] Z[0] Y[0] X[0]
FA
S[p-1] C[p-1] S[3] C[3] S[2] C[2] S[1] C[1] S[0] C[0]
CSA
(a)
(b)
(c)

Fig. 2. Compressor trees. (a) The symbol used here for a compressor tree. (b) A CSA, which is the simplest three-input compressor tree. (c) CSAs used to build a compressor tree.

technique; our optimization algorithm is independent from the type of compressor tree used and can always benefit from better compressor trees.

The typical focus in traditional high-level synthesis research has been on optimal scheduling of the dataflow graph operations and on resource sharing and binding [5]. Some works have addressed the high-level arithmetic optimization of important classes of computations (e.g., linear [6]) or, more recently, have used symbolic techniques to rewrite dataflow graphs to minimize their critical path [7]. These works have addressed arithmetic problems at a higher level, independently from—and therefore complementarily to—the selection of the most convenient arithmetic representation for the implementation.

Only a few researchers have addressed the possibility of automatically inferring the carry-save representation. Some researchers [8] have suggested to rewrite constant multiplication as addition of numbers and provided algorithms to optimize such a representation. On the other hand, Kim *et al.* [9] discuss some simple transformations of the topology of the circuit to maximize the use of cascaded carry-save adders (CSAs, see again Fig. 2)—which, in fact, constitute compressor trees. Yet, in contributions by Kim *et al.* [9], [10], the main emphasis is in the optimal allocation of the addenda to the cascaded CSAs—thus making these works somehow related to the work of Stelling *et al.* [3], albeit with a slightly different problem formulation (at bit level in the latter versus word level in the former ones). Yu *et al.* [11] also address the automatic use of a carry-save representation, but their main focus is the impact of retiming on the choice of an optimal representation.

In a contribution by Mathur and Saluja [12], they build on the work of Kim and Um and use bitwidth analysis to improve the potentials of the carry-save representation in the presence of unnecessary operand truncations. This is effectively a very particular case of the problem we address in this work (mixed arithmetic and logic operations) and, in fact, a special case that we actually do not solve with our more general approach—our work is therefore perfectly complementary to theirs.

Finally, Synopsys's synthesizers also have some capabilities to infer the use of CSAs [13]; the opportunities seized by Synopsys's behavioral optimization of arithmetic (BOA) are probably similar to those described in the work of Kim *et al.* [9]. Although somehow similar in the intents, our work is more advanced in trying to exploit more occasions for proficient uses of compressor trees. We will show that, in addition to what previous techniques addressed, we successfully optimize some new cases of practical interest. This piece of work builds on a previous contribution of ours [14], where similar results were attained; yet, the algorithm presented here warrants some novel interesting properties, and their proofs are discussed here in detail.

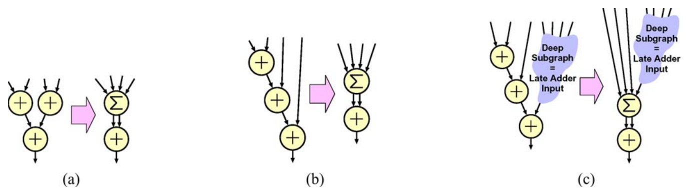


Fig. 3. Example of application of compressor trees to reduce the critical path of arithmetic circuits. In the special case of (c), using a compressor tree might actually increase the circuit delay due to the late arrival of the last addendum.

III. Arithmetic Optimizations

In essence, our goal is to transform acyclic dataflow graphs to maximize the opportunities to use compressor trees in combinatorial circuits. We call a compressor tree (see Fig. 2) a circuit which takes $N \geq 3$ input words and produces two output words S and C

$$S, C = \mathcal{F}(A_0, A_1, \ldots, A_{N-1}).$$

The function $\mathcal{F}$ can be any function such that

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

In general, but not in all cases, the use of compressor trees will reduce the critical path of the hardware implementation of a dataflow graph. Fig. 3 shows examples of graphs transformed to make use of compressor trees. It is interesting to note that the use of compressor trees makes tree-height reduction unneeded, as cases (a) and (b) indicate—or, more precisely, pushes back the problem to the implementation of optimal compressor trees (which is feasible, as described in [3], [10], [15], and [16]).

In the case of Fig. 3(c), the late arrival of an addendum can make the use of a large compressor tree counterproductive. We ignore this type of situation and implicitly assume that the use of compressor trees wherever possible is beneficial. One possible solution would be to use compressor trees only if this reduces the critical path. Yet, this solution will work for the aforementioned case but not in all cases, for example, if the original dataflow graph is like the transformed graph in Fig. 3(c). Hence, the only solution is to implement compressor trees with knowledge of the arrival times of the individual signals.

Some work has already been done on building such compressors. In the work of Oklobdzija *et al.* [4], they divide the array of input integers into vertical slices (i.e., a vertical slice consists of the i-th bit of each input for some i) and use a compressor to reduce each vertical slice. The compressor uses a three-greedy approach as a heuristic to reduce the vertical slice. In the three-greedy approach, a set I is defined for each vertical slice; it initially consists of the arrival delay of all slice input bits. In each step, the three smallest values (for example, x, y, and z) are chosen from I, and a full adder is used to produce the carry and sum of the corresponding bits. If the carry and sum bits are produced at times c and s, the three values x, y, and z are removed from I, and s is added to I while c is added to the I of

TABLE I
REWRITING RULES FOR NONPRIMITIVE ARITHMETIC OPERATIONS

$-A$	$\Rightarrow$	$\overline{A}+1$
$A-B$	$\Rightarrow$	$A+\overline{B}+1$
$A*B$	$\Rightarrow$	$\mathsf{Sum}(\mathsf{PP}(A,B))$
$A\ \langle rel\_op\rangle\ B$	$\Rightarrow$	$F_{rel\_op}(\mathsf{SGN}(A-B))$

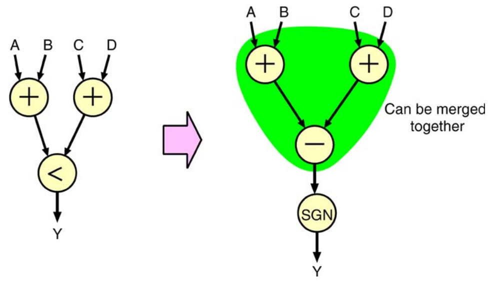


Fig. 4. Example of application where rewriting a relational comparison using the subtraction operation helps in merging two addition nodes.

vertical slice of next bit. The process stops when the size of I reduces to three elements, and the three signals are sent to a full adder. This approach is based on the assumption that the difference between the arrival times of input signals is not extremely large and that the use of compressor tree is beneficial. Such an assumption is valid for the case when the input signals are partial products of two integers but not in all cases as in Fig. 3(c). In this case, where one of the input operands arrives very late compared to other inputs, the use of an adder tree is preferred over compressor tree. To implement the compressor trees, we use the three-greedy approach with a minor difference: in our approach, we first check if two integers can be added even before the next input integer arrives; then, we first add these two integers using an adder before using the compressor tree.

A. Combining Adders

We start from an acyclic dataflow graph representing the computation to be implemented. This could be converted to a suitable hardware description language (e.g., VHDL or Verilog), and then, a logic synthesizer and a library of standard arithmetic components can be used to implement it in hardware. Instead, to obtain faster and more efficient implementations, before writing out the graph for the synthesizer, we apply three types of transformations to the graph with different goals.

1) Expose Adders in Other Arithmetic Operations: This is a rather classic set of transformations, rewriting operations such as subtractions and multiplications as logic operations and additions. The transformations are listed in Table I. The third transformation implements a parallel multiplier [2]: $\mathsf{PP}()$ produces a set of partial products, and $\mathsf{Sum}()$ represents a multiple-input addition which will be implemented in hardware using a compressor tree. The fourth transformation implements relational operator using subtraction and SGN (which returns the sign bit of an integer). As an example, given two integers A and B, checking whether B is smaller is equivalent to check whether $A-B$ is a positive integer (i.e., sign bit of $A-B$ is one). Sometimes, such rewriting also helps in merging additions, as shown in Fig. 4.

2) Rearrange Adders to Maximize Multi-Input Adders: This is the fundamental set of rules to achieve our goal: Both

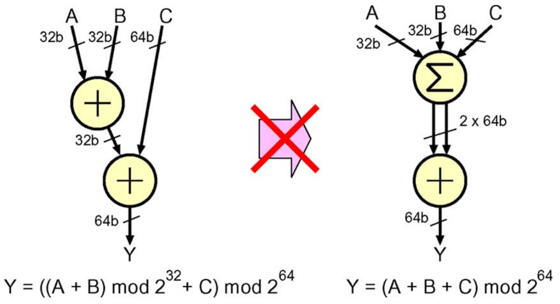


Fig. 5. Example of application where our transformations may lead to errors because of overflow.

selectors resulting from software constructs (e.g., if conversion) and other logic operations (e.g., shifts, inversions, and bitwise operations needed in programs to describe high-level operations at the bit level) are scattered among the sequences of adders exposed by the previous class of transformations. If such logic operations are not moved away, adders cannot be merged, and the opportunities for optimization will be missed. We first cluster adders by separating them from other logic operations, and then we merge them into multi-input adders. These rules are the main novelty of our approach, and conversely, their absence is the most severe limitation of other techniques discussed in Section II. These transformations are discussed at length in the following sections.

3) Carry-Save Implementation of Multi-Input Adders: This is also a simple transformation in which we replace a multi-input adder with a compressor tree followed by a single carry-propagate final adder.

Note that we do not implement any rule to optimize logic operations—we fully rely for this on the capabilities of traditional logic synthesizers. Also note that the semantic of high-level languages usually ignores overflows—e.g., $A+B$ actually represents $(A+B) \mod 2^{32}$—whereas we implement operations in full precision. If input operation graphs are derived from high-level language (as it is often the case in our examples), this could lead to errors if the programmer relied on this semantic peculiarity of the programming language. For example, in Fig. 5, the original dataflow graph computes $((A+B) \mod 2^{32}+C) \mod 2^{64}$, while the transformed dataflow graph computes $(A+B+C) \mod 2^{64}$. The two outputs will be equal if there is no overflow, but in case of overflow, the two may not be equal, and hence, our transformations lead to errors. Although rare, this may happen in applications implementing modular arithmetic—namely, cryptography. We ignore such peculiarities.

B. Sorting Problem

We formalize here the problem of grouping arithmetic operators together to improve the effectiveness of the carry-save representation. We call $G(V,E)$ the directed acyclic graph representing the dataflow of the computation to implement, typically derived from a software description and already subjected to the rewriting rules of Table I. Nodes V's represent primitive operations, and edges E's represent data dependences. The nodes of G are ordered such that if G contains an edge (u,v), then u appears before v in the ordering. The function $\mathrm{Ord}(\cdot)$ returns the position of a node in the ordering. The nodes V's

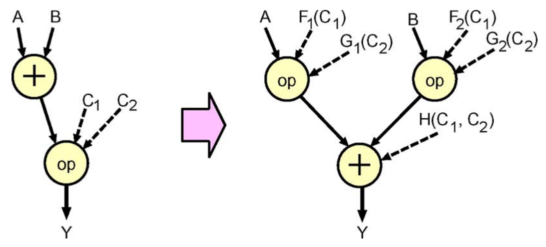


Fig. 6. General form of the sorting rules which advance logic operations over addition.

TABLE II
EXISTING POSSIBILITIES FOR ADVANCING CLASS L OPERATIONS OVER ADDITIONS

Bitwise AND, OR, and XOR	no
Bitwise negation (NOT)	yes
Right shift ($s > 0$) or left ($s < 0$)	no
Left shift ($s \geq 0$) or right ($s \leq 0$)	yes
Partial product generator	yes
Selector	yes
Equality comparison	no
SGN	no

can be of two classes: arithmetic (A) and logic (L). Arithmetic nodes are only additions, since all other arithmetic operators we consider have been rewritten using additions (see Table I). All other operators are logic nodes. The function Class$(\cdot)$ returns the class of a node. Finally, we call two graphs G and G' *semantically equivalent* if all outputs of two circuits implementing graph G and graph G', respectively, are identical under any combination of input values.

Our problem can be formalized as follows.

Problem 1: Given a graph G, transform it into a semantically equivalent graph $G'(V', E')$ where the following property holds: For all nodes u and v such that Class$(u) = \mathsf{A}$ and Class$(v) = \mathsf{L}$, it is either always Ord$(u) <$ Ord(v) or always Ord$(u) >$ Ord(v).

We call such a graph G' *sorted*. For instance, the motivational example of Fig. 1 is unsorted, whereas that of Fig. 27 is sorted and thus apt for implementation. The fact that a graph is sorted is sufficient, although not necessary, to be able to produce an optimal implementation with the use of compressor trees.

Since we have rewritten arithmetic operators like negation, subtraction, or multiplication using the rewriting rules of Table I, the only arithmetic operators in the transformed graph are additions. Hence, the aforementioned problem reduces to the following problem.

Problem 2: Given a graph G, transform it into a semantically equivalent graph $G'(V', E')$ in which no two add nodes are separated by a single logic node or a sequence of logic nodes.

To solve Problem 2, we define some graph transformations which swap the order of addition node with other logic nodes. A general form of such transformation is shown in Fig. 6. We discuss these transformations in detail in the next section.

C. Sorting Rules

Table II summarizes possible transformation rules to sort node couples and advance class L operations over additions. As it can be observed, only in some cases that such swapping is possible. As shown in Fig. 7, the following can be observed.

1) Bitwise negation (NOT) can be advanced over addition. Exchange of bitwise NOT with addition is based on the relation $-A = \overline{A} + 1$ and yields

$$\overline{\left(\sum_{i=0}^{k-1} A_i\right)} = \sum_{i=0}^{k-1} \overline{A_i} + k - 1.$$

2) Similarly, effective left shifts (i.e., left shifts with positive shift count as well as right shifts with negative shift count) can be advanced over addition, since shifting to the left is equivalent to multiplying by an appropriate power of two; this sorting rule corresponds to an elementary application of the distributive property of multiplication over addition. Formally, we can write

$$\left(\sum_{i=0}^{k-1} A_i\right) \ll S = \sum_{i=0}^{k-1} (A_i \ll S).$$

3) A similar but less elementary sorting rule consists in advancing a selector node (SEL) over addition; it is based on the existence of the identity element of addition

$$c\ ?\ \sum_{i=0}^{n-1} A_i : B = \sum_{i=0, i \neq j}^{n-1} (c\ ?\ A_i : 0) + c\ ?\ A_j : B$$

where we indicate by $c\ ?\ a : b$ the value a if the value of Boolean c is true, and b otherwise. In the transformation, j can be arbitrarily chosen. Note, however, that if B is identical to one of the A_i (for example A_k), then it is beneficial to choose $j = k$, since that leads to a degenerate selector node $c\ ?\ B : B$; in all other cases, we assign j as 0.

4) The last sorting rule consists in advancing PP over addition. This rule is based on the distributive property of multiplication over addition but is significantly more complex than other sorting rules.

$$\begin{aligned}&\mathsf{Sum}\left(\mathcal{F}\left(\mathsf{PP}(A_0 + \cdots + A_k, B)\right)\right)\\ &\qquad = \mathsf{Sum}\left(\mathcal{F}\left(\mathsf{PP}(A_0, B)\right), \ldots, \mathcal{F}\left(\mathsf{PP}(A_k, B)\right)\right)\end{aligned}$$

where $\mathcal{F}$ indicates the operators which, after prior sorting rules, may separate the PP node from its associated successor Sum (notice that both nodes were generated by rewriting multiplications as sums of partial products). Essentially, we are duplicating the PP and all $\mathcal{F}$ nodes for every addendum. Note that, unlike the other sorting rules, this one has a major cost in terms of hardware area and is sometimes not practical; hence, we need to ensure that this sorting rule is only applied when really beneficial. We will come back to this issue in Section III-J.

The last sorting rule is based on the distributivity of multiplication over addition as well as on the following property of the intermediate dataflow graph obtained by applying some instances of the aforementioned sorting rules.

Property 1: Any dataflow graph G', derived from a given dataflow graph G by applying some of the aforementioned sorting rules, possesses the following property. In G', for any

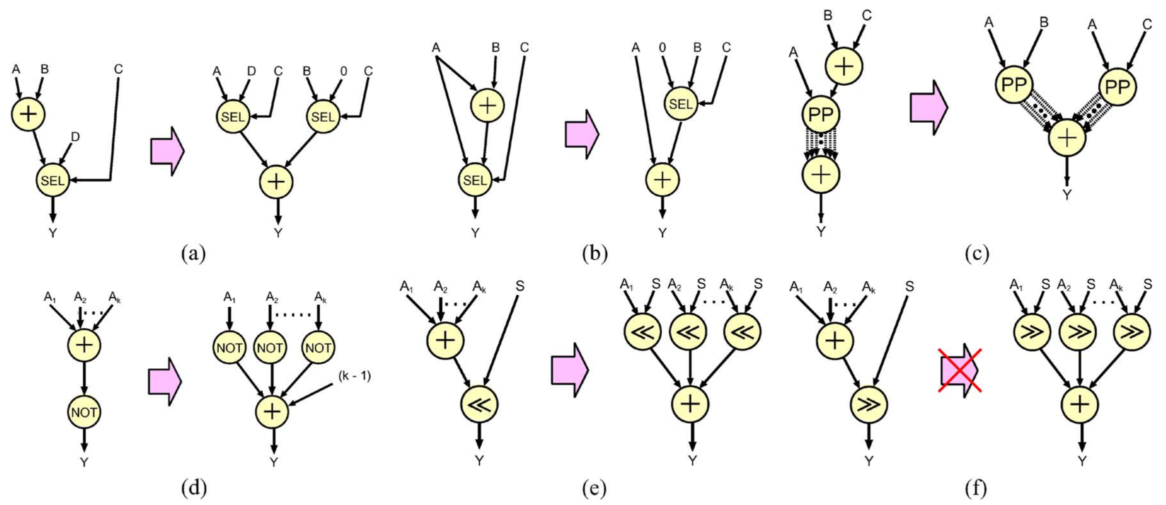


Fig. 7. Advancing (a) selector over addition in the general case, (b) selector over addition when one of the input of addition is identical to the other input of the selector, (c) partial product over addition, (d) bitwise NOT over addition, and (e) shift left over addition. Note that (f) right shift cannot be advanced over addition.

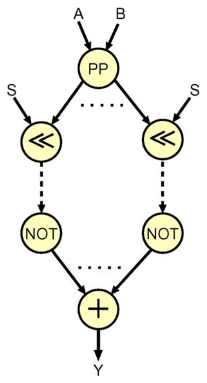


Fig. 8. All outputs of a PP node lead to an addition node via identical paths of logic nodes.

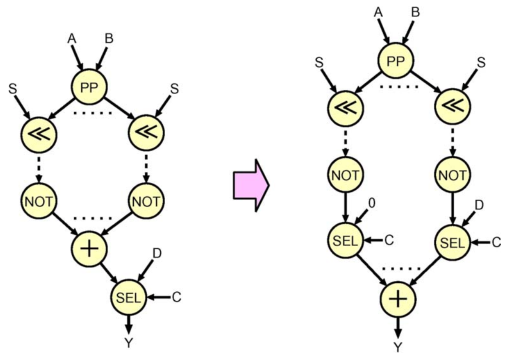


Fig. 9. Example showing that all outputs of PP lead to an addition node via identical paths after advancing a selector over an addition.

PP node, there exists an addition node such that from all outputs of the PP node, there are identical paths of logic nodes to the addition node, as shown in Fig. 8.

Proof: The proof is based on the induction on the number of sorting rules applied on G to obtain G'.

1) **Base case**: The base case is when the number of sorting rules applied is zero (i.e., G and G' are identical). In G, all the PP nodes are the nodes introduced by rewriting a multiplication node according to the third rule of Table I. In this rewriting rule, we replace a multiplication by a multioperand addition having the partial products as its inputs. It is easy to see that G satisfies the aforementioned property because all the outputs of a PP node are connected to the addition node by edges. In other words, from all outputs of PP node, there are paths of zero length to a common addition node.
2) **Induction step**: Assume that the property holds for all dataflow graphs obtained from G after applying n sorting rules. Now, let G' be the graph obtained from G after applying $n+1$ sorting rules. Consider the graph H, derived from G by applying the first n rules of the $n+1$ sorting rules. Since H is n sorting rules distant from G, according to the hypothesis, it should satisfy the aforementioned property. Now, we consider two possible cases: either the graph G' is obtained from H by applying one of the first three sorting rules (i.e., advancing NOT, shift left, or SEL over addition) or G' is obtained from H by applying the fourth sorting rule which consists of advancing a PP node over an addition node.

In the first case, where the sorting rule transforming H to G' consists of advancing a logic node L over the addition node A, if A has k input operands in H, then k copies of node L are created, each having one operand that is the same as one of the operands of node A, and all these copies have the node A as its successor. It can be seen that if there was a PP node in H such that there were paths from all outputs of the PP node to the addition node A, then there will still be paths from the output of the PP node to A, and also, they will remain identical because, in the transformation from H to G', a copy of L is added to all paths. On the other hand, if there was a PP node in H whose outputs were leading to an addition node other than A, then, in G', also all the outputs of this PP node would lead to the same addition node, because this part of graph remained unchanged in the transformation of H into G'. This shows that G' also satisfies the aforementioned property. Fig. 9 shows an example of this case when the corresponding sorting rule advances a selector over an addition.

The second case, which is shown in Fig. 10, consists of advancing a PP node P over the addition node A. Since H satisfies Property 1, in H, there will be an addition node A' such that from all outputs of P, there will be identical paths of logic

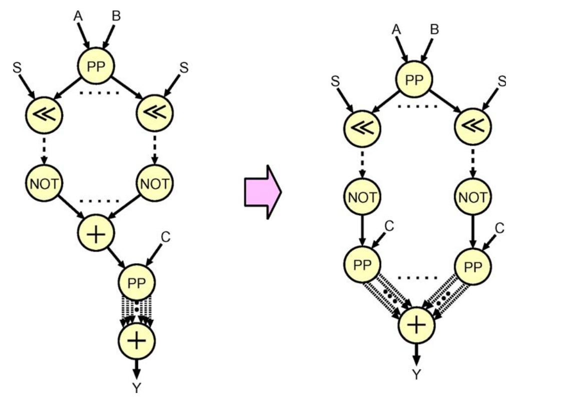


Fig. 10. Example showing that all outputs of PP lead to an addition node via identical paths after advancing PP over an addition.

nodes to A'. If A has k operands in H, then k copies of the P node are created, each having one operand which is the same as one of the operands of A and another operand which is the same as the other operand of P. All outputs of these duplicated PP nodes lead to A' by the same path, which lead the original node P to A' in H. Now, in graph H, if there was a PP node whose outputs had a path to node A, then in graph G', all outputs of that PP node will have identical paths to A' (which will be the original path in H appended with the path from P to A' in graph H). Also, the new duplicated PP nodes in G' satisfy the property, as all their outputs are connected to A' by the same path, which connected the outputs of P to A' in graph H. Hence, the transformed graph G' also has the desired property. ■

D. Additions Separated by Right Shifts

In the previous section, we have seen that if two additions are separated by left-shift operation, they can be clustered by advancing the addition over the left shift. The transformation uses the fact that the left-shift operation is equivalent to multiplication by a power of two and that multiplication is distributive over addition. Unfortunately, the same is not true for the right-shift operation. Right shift is equivalent to integer division, which is not distributive over addition. However, one can observe this simple property

$$\left\lfloor \frac{A+B}{2^k} \right\rfloor = \left\lfloor \frac{A}{2^k} \right\rfloor + \left\lfloor \frac{B}{2^k} \right\rfloor + \left\lfloor \frac{(A \bmod 2^k) + (A \bmod 2^k)}{2^k} \right\rfloor, \text{ i.e.,}$$

$$\left\lfloor \frac{A+B}{2^k} \right\rfloor = \left\lfloor \frac{A}{2^k} \right\rfloor + \left\lfloor \frac{B}{2^k} \right\rfloor + \mathsf{CGEN}\left((A \bmod 2^k) + (A \bmod 2^k)\right).$$

Here, CGEN corresponds to carry generating function, which produces a single bit representing the carry output of the addition of the two inputs. In other words, advancing addition over right shift produces an output which can be corrected by adding a correction bit in the produced output. This transformation is shown in Fig. 11.

Note that this transformation may increase the critical path delay marginally because of the new instruction CGEN. The initial delay was $D_{\text{add}}(n)$, where n is the bitwidth of the operands; however, the delay after this transformation is $D_{\text{add}}(n-k) + \mathsf{CGEN}(k)$. Fig. 12 shows the delays of the two implementations

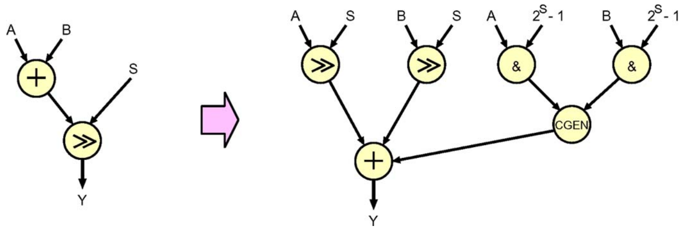


Fig. 11. Advancing an addition over right shift can be corrected using a correction bit.

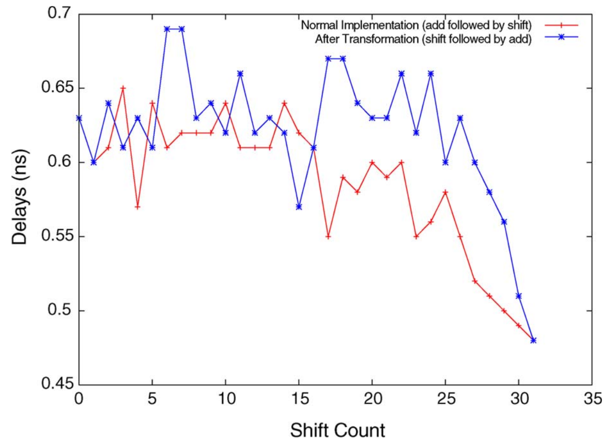


Fig. 12. Transformation of advancing addition over right shift increases the delay, but not significantly. The corresponding circuit is the addition of two 32-b integers right shifted by some constant.

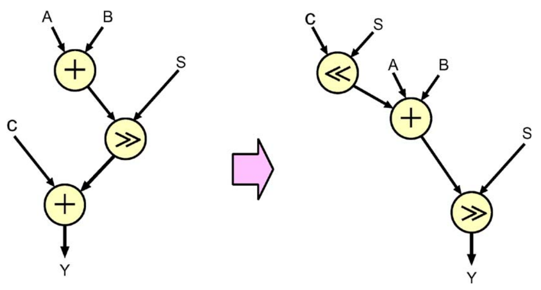


Fig. 13. Another way of merging addition separated by a right-shift operation.

of addition followed by right shift. The first implementation is the normal implementation, i.e., addition followed by right shift, and the second implementation corresponds to the implementation after advancing addition over right shift. It is evident from Fig. 12 that the transformation of advancing addition over right shift deteriorates the performance of the circuit slightly. However, if this transformation helps in clustering two addition nodes, the overall delay of the circuit will reduce. This is because by clustering two additions together, we save the delay of an adder, which is significantly higher than the delay penalty caused by advancing addition over right shift (particularly for smaller shift counts).

Another way of merging additions separated by right-shift operation is shown in Fig. 13. This transformation is based on the following identity:

$$\left\lfloor \frac{A+B}{2^k} \right\rfloor + C = \left\lfloor \frac{A+B+2^k C}{2^k} \right\rfloor.$$

The advantage of this transformation over the previous one is that it merges the complete n-bit additions, while previous

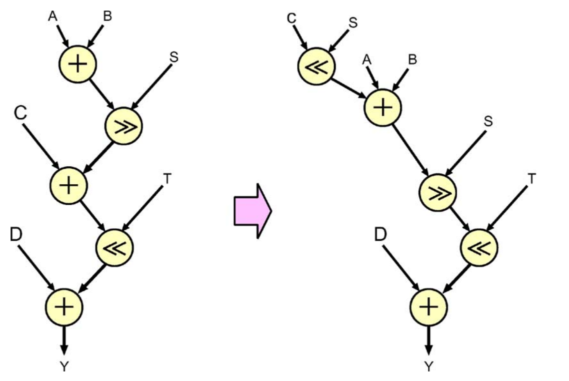


Fig. 14. Since right shift cannot be advanced over left-shift operation, the additions separated by right shift and left shifts cannot be merged via the transformation of advancing right shift over addition.

transformation merges only the additions of most significant $n - k$ bits. On the other hand, the disadvantage of this transformation is that, unlike the other transformations, this transformation advances right shift over addition. This means that if right shift cannot be advanced over some logic nodes, then additions separated by a sequence of right shift and that logic node cannot be merged via this transformation. One such example is shown in Fig. 14, where additions are separated by a right-shift and a left-shift operation. After advancing right shift over addition, we get left shift and right shift as neighboring nodes in the dataflow graph. Since these two nodes cannot be swapped, the additions separated by these nodes cannot be merged. In our algorithm, we use the second transformation when the sequence of logic nodes separating additions consists of only right-shift operation; otherwise, we use the previous transformation of advancing addition over right shift.

Since any of the two transformations is not confluent with other transformations, the final result depends on the order in which we apply this transformation. Since this transformation is less beneficial to other transformations, as well as it violates the rules of the reduction system presented in the next section, we apply this transformation at the very end when no other transformations are applicable.

E. Redundancy of Some Sorting Rules

We consider only sorting rules to advance logic operations over additions. It might be possible that there are some sorting rules which cluster additions by advancing addition over logic nodes. As an example, the following sorting rule advances addition over selector node, and as shown in Fig. 15, this sorting rule helps in clustering additions:

$$X + (c\,?\,Y : Z) = c\,?\,(X + Y) : (X + Z).$$

Still, we can ignore these sorting rules as the next theorem states the redundancy of such sorting rules.

Theorem 1: If a sequence of swapping operations, which includes at least one advancing an addition over a selector node, results in clustering two addition nodes A_1 and A_2, then the two addition nodes can also be clustered by applying a sequence of swapping sorting rules, none of which advances addition over selector. Hence, the rule to advance addition over a selector is redundant and can be ignored (Fig. 15).

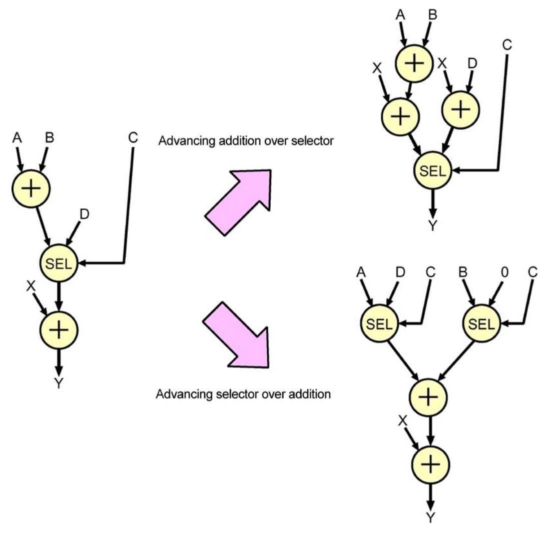

A
B
C
D
SEL
X
Y
Advancing addition over selector
A
B
X
X
D
C
SEL
Y
Advancing selector over addition
A
D
C
B
0
C
SEL
SEL
X
Y

Fig. 15. Example showing that the sorting rule advancing addition over selector is redundant and does not cluster any additional set of add nodes.

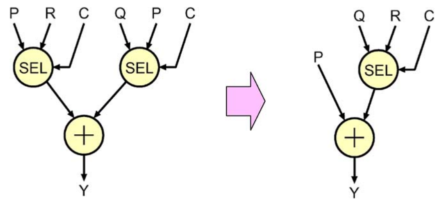


Fig. 16. Merging two selector nodes into a single selector.

Proof: Let $T_1, T_2, \ldots, T_m$ be the sequence of sorting rules which clusters two addition nodes A_1 and A_2. Consider that $T_{r_1}, T_{r_2}, \ldots, T_{r_k}$ are those sorting rules which advance addition over selector $(\{r_1, r_2, \ldots, r_k\} \subset \{1, 2, \ldots, m\})$. Since these sorting rules advance additions over selector nodes, they will not enable the application of nontrivial sorting rules advancing a logic operation over additions (of course, they will enable sorting rules advancing the same selector over addition, which is just the inverse of T_{r_i}'s). Therefore, we can apply all other sorting rules except T_{r_i}'s because none of them are enabled by T_{r_i}'s. Also, by the persistence property which we will prove in Thorem 2, we can see that T_{r_i}'s will still be applicable. Since, after applying a sequence of sorting rules advancing addition over selector, we can merge the two addition nodes A_1 and A_2, it implies that there is a sequence of only selector nodes in the path from A_1 to A_2. Now, instead of advancing A_1 over the selectors, we advance the selectors over A_2 and still cluster the two addition nodes. Therefore, the sorting rule advancing addition over selector is redundant. ■

Although this only proves the redundancy of the sorting rule advancing addition over a selector node, it easily generalizes to other logic operations.

F. Clean-Up Transformations

Along with the aforementioned sorting rules, we need to use some clean-up transformations, too. Albeit most of these transformations are trivial and consist of propagating constants, suppressing degenerate compressor trees, etc., some of them are nontrivial as the one shown in Fig. 16. This transformation merges two selector nodes. It can be noted that if we are

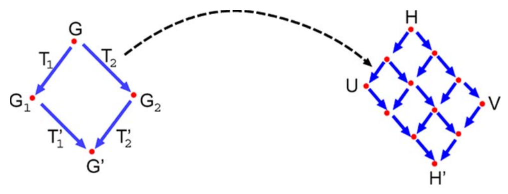


Fig. 17. Local confluence and the other properties imply the convergence of the sorting rules.

applying this transformation after applying any of the regular sorting rules, then we need not to consider the special case of the sorting rule advancing a selector over addition where the addition node and the selector node share a common input.

G. Order Independence

Sorting would be a feasible task if it were always possible to exchange the order of class A and L nodes without changing the semantics of the dataflow graph. Yet, in many cases, no sorting is possible: sorting rules are not available for basic bitwise logic operations nor for equality comparisons (i.e., a bitwise XOR and an AND reduction operation). One could then apply any of the existing sorting rules (excluding the rule of advancing addition over right shift) on the graph wherever possible to cluster additions as much as possible. The question of what is the most appropriate order in which to apply the sorting rules naturally arises. In fact, the sorting rules constitute a finitely terminating and confluent (i.e., convergent) reduction system [17], and thus, the order is immaterial to the result.

Property 2: Given a dataflow graph G, any possible sequence of sorting rules resulting in a graph which cannot be further transformed results in the same final graph G'.

The proof depends on the following properties.

1) **Topology independence**: All sorting rules are independent of the topology of the graph—e.g., we can always advance left shifts over additions irrespective of the topological structure of the graph.
2) **Persistence**: All sorting rules are such that they preserve the applicability of all other transformations. That is, the application of any sorting rule does not transform the graph in such a way that a sorting rule which was applicable before becomes inapplicable.
3) **Local confluence**: It does not matter in which order we apply the two sorting rules on a given graph, the final resulting graph will be the same.

The proof of Property 2 is based on local confluence and uses it recursively to prove global confluence (Fig. 17). The possible graphs obtained with the sorting rules form a lattice. Hence, in order to complete the proof, we only need to prove the persistence and local confluence properties of the sorting rules.

Theorem 2: Persistence—Given a graph G, if there is a sorting rule T_2 applicable before applying the sorting rule T_1, then it is also applicable after applying T_1.

Proof: The proof consists in enumerating all possible combinations of T_1 and T_2. Here, we will give the proof only for one combination because, for other combinations, it can be proved similarly. The combination that we consider is when T_1 consists of advancing a selector node SEL_1 over an addition node ADD_1 and T_2 consists of advancing a selector node SEL_2 over an addition node ADD_2. Now, there can be four possibilities depending on the relative topological positions of SEL_1 and SEL_2. Fig. 18 shows these four possibilities. In the figure, SEL_1 is represented by continuous lines and SEL_2 is represented by dotted lines. We consider each of these possibilities individually.

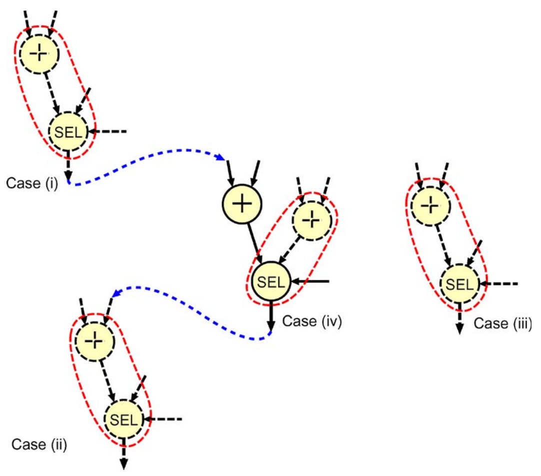


Fig. 18. Example illustrating the persistence of transformations. The first transformation corresponds to swapping the order of SEL and add node connected by solid lines. The second transformation corresponds to swapping the order of SEL and add node connected by dotted lines. The four cases correspond to the relative topological order of the nodes where the two transformations are applied.

1) SEL_2 **precedes** SEL_1 **topologically or vice versa**: We will consider the case where SEL_2 precedes SEL_1 topologically; the other one can be proved in a similar way. In this case, which is shown as case 1 in Fig. 18, it can be seen that even after advancing SEL_1 over ADD_1, the topological order between SEL_2 and ADD_2 will remain the same (i.e., still SEL_2 is the successor of ADD_2). Therefore, we can still advance SEL_2 over ADD_2, which implies the applicability of T_2 after applying T_1.
2) **No topological order between** SEL_1 **and** SEL_2: Similarly, in the case where there is no topological order between SEL_1 and SEL_2 (shown as case 3 in Fig. 18), there will be no change in topological order between SEL_2 and ADD_2 after applying T_1, and, hence, persistence of transformation holds.
3) $\mathsf{SEL}_1 = \mathsf{SEL}_2$: In this case, after advancing SEL_1 over ADD_1, SEL_1 will be duplicated several times, and one of these duplicate SEL nodes will have ADD_2 as its child, which means that, still, we can advance SEL_2 over ADD_2; in other words, T_2 is still applicable. ■

Theorem 3: Local confluence—Given a graph G and two applicable sorting rules T_1 and T_2, whose application results in graphs G_1 and G_2, respectively, there exist two sorting rules T'_1 and T'_2 such that by applying T'_1 on graph G_1 and T'_2 on graph G_2, one obtains the same graph G'.

Proof: The proof of local confluence is similar to the proof of persistence and is performed by considering all possible combinations of T_1 and T_2. Here, we will consider one of these combinations because for other combinations, it can be proved similarly. The combination that we consider is when T_1 consists of advancing a selector node SEL_1 over an addition node ADD_1 and T_2 consists of advancing a selector node SEL_2 over an addition node ADD_2. Now, there can be four possibilities depending on the relative topological positions of SEL_1 and SEL_2 (as in the proof of Theorem 2). We consider each of these cases individually.

1) SEL_2 **precedes** SEL_1 **topologically or vice versa**: We will consider the case where SEL_2 precedes SEL_1 topologically; the other one can be proved in a similar way. In this case, the two sorting rules are totally independent of each other; therefore, no matter in which order we apply T_1 and T_2 (due to persistence of transformations, each of them will be applicable after applying the other transformation), they will result in the same final graph. Therefore, $T_1' = T_2$ and $T_2' = T_1$.
2) **No topological order between** SEL_1 **and** SEL_2: Similarly, in the case where there in no topological order between SEL_1 and SEL_2, the theorem holds true because in this case, also the two sorting rules are independent of each other and do not change the part of the graph modified by the other sorting rule.
3) $\mathsf{SEL}_1 = \mathsf{SEL}_2$: In this case, too, we can see that $T_1' = T_2$ and $T_2' = T_1$, as shown in Fig. 19. More formally, we can write

$$\begin{aligned}
G &= c\ ?\ \sum_{i=0}^{m-1} A_i : \sum_{j=0}^{n-1} B_j,\\
G_1 &= c\ ?\ A_0 : \sum_{j=0}^{n-1} B_j + \sum_{i=1}^{m-1} (c\ ?\ A_i : 0),\\
G_2 &= c\ ?\ \sum_{i=0}^{m-1} A_i : B_0 + \sum_{j=1}^{n-1} (c\ ?\ 0 : B_j), \text{ and}\\
G' &= c\ ?\ A_0 : B_0 + \sum_{i=1}^{m-1} (c\ ?\ A_i : 0) + \sum_{j=1}^{n-1} (c\ ?\ 0 : B_j).
\end{aligned}$$

■

H. Useful Mobility

The next issue is that our sorting rules do not necessarily reduce the critical path delay: Except for the fourth sorting rule (advancing PP over addition), all other sorting rules reduce the critical path exclusively if they result in merging two addition nodes. Some of these sorting rules have no cost in terms of critical path delay or hardware area (e.g., advancing left shifts over addition), while others have some associated cost. For instance, advancing bitwise NOT over addition increases the critical path delay marginally because it increases the number of inputs of the addition node by one. Similarly, advancing a selector over an addition has a cost in terms of area. Therefore, one would like to use these sorting rules only if they will result in merging at least two addition nodes. In this section, we discuss how to identify the logical nodes which will result in merging two additions if they are advanced over the preceding addition.

We call *movable logic nodes* the nodes which can be advanced over addition using some sorting rule. As discussed in the section describing sorting rules, four types of nodes are movable: left shifts, bitwise NOTs, selectors, and partial products. We call *useful movable logic nodes* those movable nodes that are susceptible to bringing an advantage if a sorting

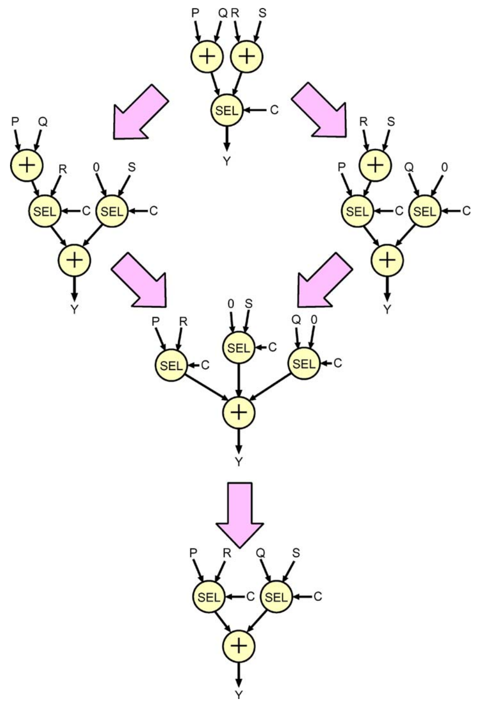


Fig. 19. Example of local confluence: Whatever the order followed in applying two sorting rules, the result is the same.

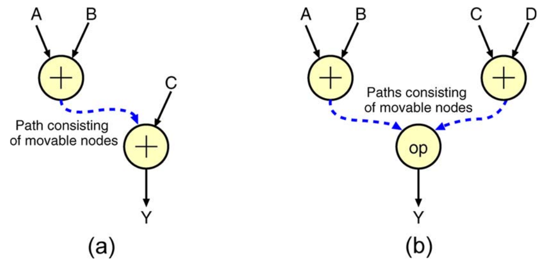

A
B
C
D
Path consisting of movable nodes
Paths consisting of movable nodes
op
Y
(a)
(b)

Fig. 20. All the nodes in the path are useful mobile nodes.

rule is applied to them. We mark movable nodes as useful if any of the following cases holds true.

1) They lie on a path between two additions, and such a path consists exclusively of movable nodes [Fig. 20(a)].
2) They lie on one of two paths connecting two addition nodes with a single movable node, and both paths consist exclusively of movable nodes. The joining movable node is also a useful movable node [Fig. 20(b)].

Note that we do not consider paths going through the shift count edge of the left-shift operator and the selector control edge. Clearly, the aforementioned two conditions are necessary for the two additions to merge if appropriate sorting rules are applied. The conditions are not sufficient, because some of the movable nodes on the paths might not be swapped; this can be due to the fact that some movable nodes have more than one successor. One might thus be tempted to call useful movable the nodes on a path if all the nodes in the path have exactly one successor; however, even nodes with more than one successor can be usefully removed from a path as shown in Fig. 21.

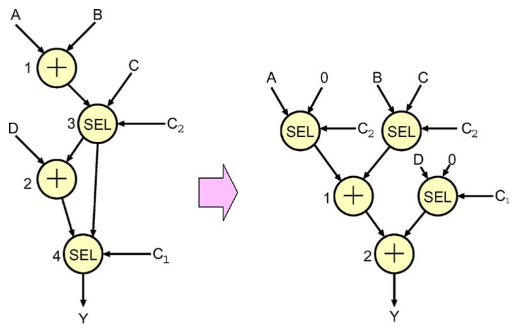


Fig. 21. Example showing that nodes with more than one successor can also be useful movable nodes.

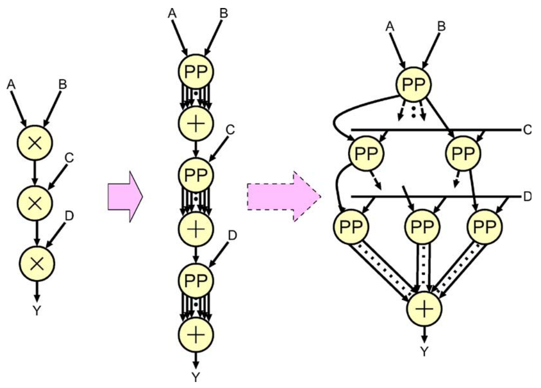


Fig. 22. Huge compressor tree resulting from advancing addition over partial product in the computation of $A \times B \times C \times D$.

I. Compressing the Multi-Input Addition Followed by Partial-Product Generator

In the earlier sections, we have seen that in almost all the instances of advancing addition over logic node, the hardware area of the corresponding circuit increases due to duplication of logic nodes. The situation becomes worse in the case of advancing addition over partial products. The main reason for this is the large number of operands of the summation following partial-product generation. For example, the multiplication of two n-bit integers is written as

$$A \times B = \mathrm{pp}_1 + \mathrm{pp}_2 + \cdots + \mathrm{pp}_n.$$

Furthermore, if we have to compute $A \times B \times C$, it will be equivalent to

$$A \times B \times C = \sum \mathsf{PP}(\mathrm{pp}_1 + \mathrm{pp}_2 + \cdots + \mathrm{pp}_n, C).$$

According to the transformation of advancing addition over partial products, we can write it as

$$A \times B \times C = \sum \mathsf{PP}(\mathrm{pp}_1, C) + \sum \mathsf{PP}(\mathrm{pp}_2, C) + \cdots + \sum \mathsf{PP}(\mathrm{pp}_n, C).$$

Note that this results in a compressor tree with n^2 input integers; however, if we do not advance the addition over partial products, then we need two compressor trees: one to compute $A \times B$ and the other to compute $(A \times B) \times C$. Each of these two compressor trees has n inputs. Even worse situation is shown in Fig. 22 where we compute the product of four integers.

This means that, although by advancing addition over partial product we save the delay of a final adder, we pay a huge cost in terms of hardware area due to a huge compressor tree. Here, we present a way to reduce the area penalty without sacrificing the delay gain. Whenever we have an addition followed by partial-product operation, before advancing addition over partial product, we compress the addenda of the addition into two summands C and S using a compressor tree. This way, we still save the delay of final adder, and also, the number of inputs of the final compressor tree does not grow exponentially. In other words, we have partitioned the large compressor tree into two small compressor trees. Although the delay of larger compressor can be smaller than the cumulative delays of the two smaller compressor trees, the differences are only marginal due to the equality

$$\mathsf{log}(mn) = \mathsf{log}(m) + \mathsf{log}(n).$$

The delay of a compressor tree is $O(\mathsf{log}k)$, where k is the number of input integers to the compressor tree. In the aforementioned discussion, the number of inputs to the large compressor is n^2, while the number of inputs to the smaller compressor trees is n. It is evident that the difference in the delays of the two implementations is only marginal. However, the area of a compressor tree is $O(kw)$, where k is the number of inputs and w is the bitwidth of inputs. This means that the cumulative area of smaller compressor trees is significantly smaller than the area of larger compressor tree.

J. Area-Timing Design Space Exploration

The last issue is that there might be some sorting rules which will result in merging addition nodes but may have a large area cost associated with them. Such sorting rules are impractical: The rule to advance PP over addition is a typical case because it may grow prohibitively the size of the following adders. Therefore, we will need to explore the design space for all Pareto-optimal solutions, that is, solutions that are either better in area or in critical-path delay than any other solution.

Fig. 23 shows an algorithm to generate all Pareto-optimal solutions. The algorithm is conservative in that it does not guarantee that all generated solutions are Pareto optimal, but certainly, it finds all the Pareto-optimal solutions. The algorithm consists of three steps: 1) In the first step, we rewrite operations according to the rules of Table I in order to expose adders as much as possible; 2) in the second step, we mark logic nodes as useful movable nodes if swapping them with an addition node will result in clustering additions; and 3) in the third step, which is the crucial one, we generate all Pareto graphs. Fig. 24 graphically suggests this last step. Essentially, we split the sorting rules into two classes: those which increase the area and those which do not. The idea is to apply each class of rules independently: While further area-constant rules are applicable, graphs cannot be Pareto-optimal solutions. Once the area-constant rules are not applicable anymore, solutions are potentially Pareto optimal ($G_1, G_2, \ldots, G'$ in Fig. 24). Then, all area-increasing rules can be applied, and the process reiterated until the final graph G' is reached. In this way, we

```
cleanUp (Graph) {
  do {
    change = false;
    changes |= propagateConstants(Graph);
    changes |= groupConstantsInCompressors(Graph);
    changes |= mergeShiftsByConstant(Graph);
    changes |= mergeSelIfPossible(Graph);
    changes |= suppressUselessCompressors(Graph);
  } while (changes);
  updatePrecisionAndPrunePartialProductEdges(Graph); }

appendGraph($\mathcal{L}$, Graph) {
  Graph g;
  g = introduceCompressor(Graph);
  $\mathcal{L}$.append(g); }

applyCostlessTransformations (Graph) {
  do {
    change = false;
    changes |= advanceNotOverAdd(Graph);
    changes |= advanceShiftOverAdd(Graph);
    changes |= mergeAddWithAdd(Graph);
    cleanUp(Graph);
  } while (changes); }

genPareto ($\mathcal{L}$, Graph) {
  for all costly transformations $T$ {
    applyTransformation(Graph, $T$);
    applyCostlessTransformations(Graph);
    appendGraph($\mathcal{L}$, Graph);
    genPareto($\mathcal{L}$, Graph); } }

optimise (Graph) {
  rewriteSubtractions(Graph);
  rewriteNegations(Graph);
  rewriteMultiplications(Graph);

  markUsefulMobile(Graph);

  // create list with all Pareto optimal solutions
  List $\mathcal{L}$;
  applyCostlessTransformations(Graph);
  appendGraph($\mathcal{L}$, Graph);
  genPareto($\mathcal{L}$, Graph);
  output($\mathcal{L}$); }
```

Fig. 23. Optimization algorithm.

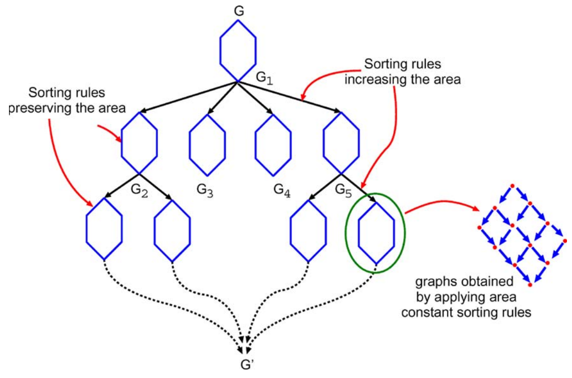

G
G_1
Sorting rules increasing the area
Sorting rules preserving the area
G_2
G_3
G_4
G_5
graphs obtained by applying area constant sorting rules
G'

Fig. 24. Finding all Pareto solutions without generating all graphs.

can generate all graphs susceptible of being Pareto optimal without generating all solutions. We have also noted that the number of Pareto-optimal graphs is not too many, and hence, it is not impractical to generate all of them. Table III shows the size and number of Pareto-optimal graphs generated for some benchmarks.

One can see that in the function *applyCostlessTransformations()*, we are also doing some clean-up transformations, as mentioned in Section III-F. At this stage, we also perform bitwidth analysis for the reasons outlined in the next section.

TABLE III
NUMBER OF PARETO-OPTIMAL GRAPHS GENERATED FOR SOME BENCHMARKS

Benchmark	Number of Nodes	Number of Pareto-Optimal Graphs
ADPCM Decoder	30	17
G721	103	2
Shift-and-add 8-bit Multiplier	94	2
Multiply Accumulate ($a \times b + c$)	7	2
Polynomial of 4th Degree (Optimised)	19	6
Polynomial of 4th Degree (Horner)	16	11
Video Mixer	85	14
H-264	17	4
FIR	31	127

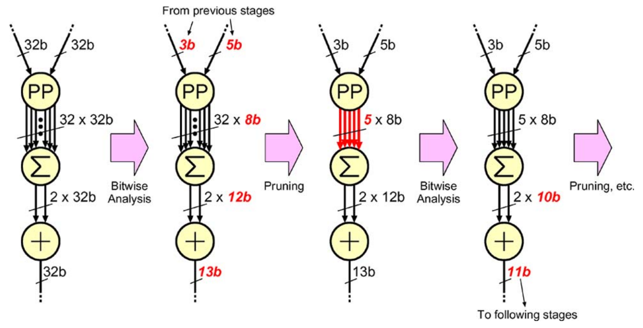


Fig. 25. Example of application of the bitwidth analysis to prune a graph.

K. Bitwidth Analysis

Implementing application-specific instructions is beneficial, among other reasons, because the computation can be implemented on a reduced number of bits if full precision is not needed—operators are therefore faster and smaller, and consume less energy.

In principle, it would seem that bitwidth analysis could be left completely to the logic synthesizer and to the arithmetic component generator producing adders, multipliers, etc. In other words, the number of significant bits representing each value does not appear to have an impact on the dataflow graph topology; hence, it seems irrelevant for the optimizations described in the previous sections. Yet, this is not entirely true, and there is a single case of precision affecting the topology. To avoid inferior results, this needs to be taken care of before passing the result to a synthesizer: the bitwidth of the factors entering a PP node influences its output count (equal to the bitwidth of one of the two factors) and, consequently, the number of inputs of the successive compressor trees.

Because of this, we need to analyze the number of bits required for every variable in the dataflow graph. To obtain this information, we have implemented a bitwidth analysis, which is very similar to the algorithms described by Mahlke *et al.* [18] and probably similar to the analysis that the synthesizer we use implements internally. We use this bitwidth analysis pass as follows: During optimization, we assume full precision everywhere in the graph (32 bits in our case) and apply all rewriting rules (notably the multiplication rewriting rule, Table I) accordingly. Then, once the optimization algorithm described in the previous section has been run, we apply the simplification loop shown in Fig. 25 to prune the graph from unnecessary edges. Multiple iterations are needed until convergence is attained, because the reduction of input edges in compressors may reduce the precision of their outputs. These, in turn, trigger further pruning later in the graph.

TABLE IV
RESULTS OF OUR ARITHMETIC OPTIMIZATIONS ON SOME BENCHMARKS

Benchmark	Original		Synopsys BOA [13]		**Our Arithmetic Optimisations**			
					Min Delay		**Min Area**	
	Delay (ns)	Area (mm^2)	Delay (ns)	Area (mm^2)	**Delay (ns)**	**Area (mm^2)**	**Delay (ns)**	**Area (mm^2)**
ADPCM Decoder [1]	1.92	.015	1.92 (0%)	.015	1.04 **(-46%)**	.013 **(-14%)**	1.07 **(-44%)**	.012 **(-23%)**
manually optimised	1.18	.010						
G.721 [1]	4.92	.077	4.72 (-4%)	.086	3.65 **(-26%)**	.064 **(-17%)**	3.65 **(-26%)**	.064 **(-17%)**
Shift-and-add 8-bit Multiplier	2.31	.017	2.31 (0%)	.017	1.48 **(-36%)**	.016 **(-2%)**	1.49 **(-36%)**	.015 **(-2%)**
manually optimised	1.63	.009						
Multiply Accumulate ($a \times b + c$)	2.02	.015	1.47 (-27%)	.014	1.60 **(-21%)**	.020 **(+33%)**	2.02 **(0%)**	.015 **(0%)**
Polynomial of 4th Degree (Optimised)	5.22	.130	5.64 (+8%)	.144	5.18 **(-0.8%)**	.328 **(+151%)**	5.22 **(0%)**	.131 **(0%)**
Polynomial of 4th Degree (Horner)	6.67	.096	6.43 (-4%)	.104	5.32 **(-23%)**	.358 **(+273%)**	6.67 **(0%)**	.096 **(0%)**
H-264 VBSME	2.37	.050	2.37 (0%)	.050	2.20 **(-7%)**	.049 **(-2%)**	2.20 **(-7%)**	.049 **(-2%)**
FIR Filter Kernel	5.66	.370	5.66	.370	3.39 **(-40%)**	.501 **(+35%)**	5.66 **(0%)**	.370 **(0%)**
Video Mixer [13]	4.88	.206	4.46 (-9%)	.143	4.45 **(-9%)**	.229 **(+14%)**	4.65 **(-5%)**	.196 **(-6%)**

IV. EXPERIMENTAL RESULTS

We have written an experimental optimizer which takes dataflow graphs (possibly extracted automatically from application code written in C) and returns a set of graphs appropriate for hardware implementation. Both the original and the optimized graphs are converted to VHDL and synthesized using a recent version of Synopsys's *Design Compiler* with the arithmetic generators of the *DesignWare* library. For the optimized graphs, we implemented two special netlist generators for partial-product generators (essentially AND networks) and for compressor trees. The latter are built with a technique similar to the already mentioned three-greedy approach [3].

We have run several benchmarks through our optimizer pass, and we have synthesized the results for an ASIC library of a common 0.18-μm CMOS technology. The results are shown in Table IV. The "Original" columns refer to the direct synthesis of the dataflow graph. "Synopsys BOA" uses the optimizations available in some versions of Synopsy's *Design Compiler* and in *Behavioral Compiler* as mentioned in the related work section. The results of the columns "Our Arithmetic Optimisations" are obtained with the algorithm in Fig. 23. Note that all compressor trees are optimized using the modified version of the three-greedy approach, as discussed in Section III. Here, we have two sets of columns. The first columns show the Pareto solution with minimum delay, and the second columns show the Pareto solution with minimal area. For two designs, under the "Original" columns, we have reported the results of manual optimizations; they imply the presence of a designer who fully understands the arithmetic meaning of the dataflow graph and who implements it with the best components available from a library.

The first three circuits in Table IV are typical arithmetic dataflow graphs coming from software descriptions. The first is the example of Fig. 1. The second (G.721) is a kernel of a standard audio encoder [1]. It represents an *ad-hoc* limited-precision floating-point multiplier including a small quantization table. The last is a classic shift-and-add multiplier, as one could write it for a processor missing a native multiplication instruction. Functionally, it has some similarity to the ADPCM example but is, in fact, coded in C in a profoundly different manner: It does not use if−then constructs, and hence, the

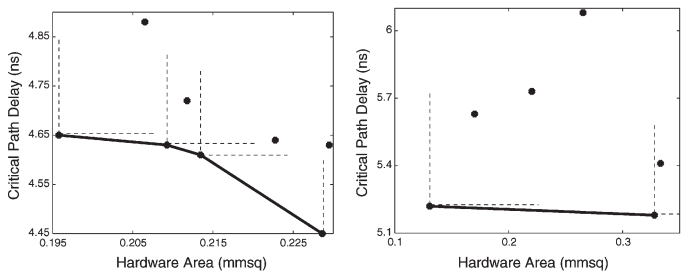


Fig. 26. Solutions reported by our algorithm for the (left) video mixer and (right) polynomial of fourth degree (optimized).

dataflow graph does not contain SEL nodes. The bottom six benchmarks are simple purely-arithmetic circuits: the first is a multiplication followed by an addition, the second and the third- are fourth-degree polynomials written in two different forms, the fourth and fifth are the small kernels inside H.264 and finite-impulse response (FIR) filter, and the last is a video mixer application that is appropriate for demonstrating classic arithmetic optimizations [13].

In Table IV, reading the results from the bottom group, one can see that on some arithmetic benchmarks, our optimization techniques achieve practically the same results of existing optimizers. In the cases of the first group, where logic operations are scattered among arithmetic computations, BOA misses completely the potentials for optimization as would the other algorithms described in the literature. Our algorithm, on the other hand, is effective in separating logic and arithmetic operations and, in some cases, manages to sort completely the dataflow graph, making an optimal implementation possible. Once we have optimized the architecture of the arithmetic part, we then rely on the synthesizer for an effective simplification of the logic network. Timingwise, our results have critical paths reduced by up to 46% and are even marginally better than the reference manual implementations—probably due to a better quality of the generated compressor trees compared to the *DesignWare* components. On these circuits, we also have small area advantages, in the range of 10%–20%.

For two benchmarks, we plotted on a delay–area graph the few potential Pareto-optimal solutions reported by our algorithm (Fig. 26). Notice that all Pareto-optimal solutions

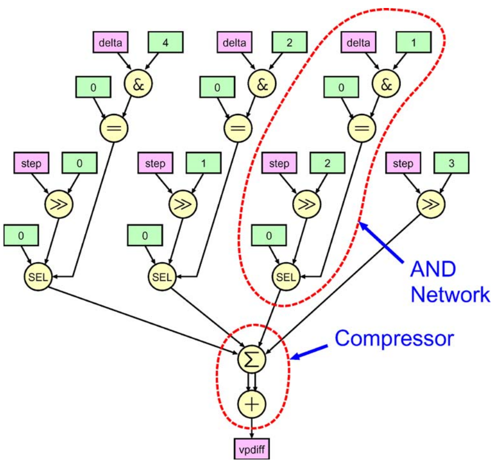


Fig. 27. Motivational example of Fig. 1 after optimization is fully sorted. The implementation is optimal, as suggested also by Table IV (ADPCM decoder).

are guaranteed to be included in the solutions reported by our algorithm. The graphs show that video mixer offers a tradeoff between delay and area, although all solutions are significantly close to each other. The polynomial of fourth degree shows one Pareto-optimal solution which is only marginally better in delay at the cost of a very significant area penalty. Although, in general, the design space appears to be relatively limited, it is important that our algorithm can apply all sorting rules, irrespective of their cost, and generate all solutions: Some of the area expensive rules would be prohibitive otherwise, although in some cases, they could be fundamental in achieving significant delay reduction. As an example, Fig. 27 shows the fastest solution corresponding to the graph in Fig. 1. Once passed to a synthesizer, each complex branch—made only of L nodes and corresponding to a partial-product generation network—gets simplified as expected into a set of simple AND gates.

Another notable example is the kernel of H.264 VBSME (variable block size motion estimator), which consists of a SAD network, that computes the sum of absolute differences. There are various ways of computing absolute difference; the most naive way is

$$X = (A > B)\ ?\ A - B : B - A.$$

This can be implemented using a multiplexer (i.e., SEL node), as shown in Fig. 28. When we take the sum of many absolute differences, the inner sums in the computation of absolute differences cannot be clustered due to the SEL node. However, when we apply our transformations on the naive implementation of absolute difference value, it advances addition over SEL and results in an implementation shown in Fig. 28. In fact, this implementation is the same as the one mentioned in [19] once we rewrite the redundant SEL operations using simpler gates based on the following identity:

$$c\ ?\ \bar{A} : A = c \oplus A.$$

Since, in this implementation, addition is the final operation in the sum of many absolute differences, these sums can be clustered and a bigger compressor tree can be used.

Note that, in general, this way of computing the absolute value is not beneficial as it increases the critical path delay due to the additional comparator in critical path. However, the delay of a comparator, which is equivalent to carry generation, is smaller than the delay of the complete adder, as shown in Fig. 29. This means that if this transformation helps in clustering two additions, it will save the delay of an addition which is more than the penalty caused by the additional comparator in critical path, i.e., the overall delay of the circuit will decrease.

The kernel of an eight-tap FIR filter is another example which benefits significantly from our transformations. As shown in Fig. 30, the initial dataflow graph consists of additions separated by right-shift operations. When we apply the transformation of advancing right shift over addition, as described in Section III-D, the resulting dataflow graph has all additions clustered and can be implemented using a compressor tree. The transformed graph has 40% smaller delay compared to the original implementation.

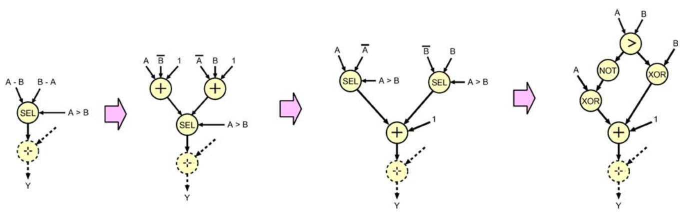


Fig. 28. Our transformation can convert a naive implementation of absolute difference to the one discovered manually and appropriate for SAD computation.

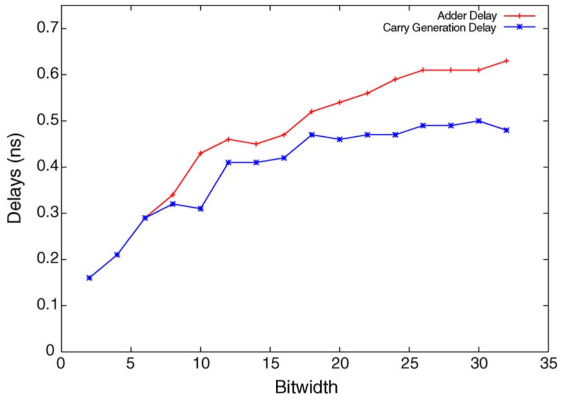


Fig. 29. Circuit for carry generation is faster than the circuit for complete adder.

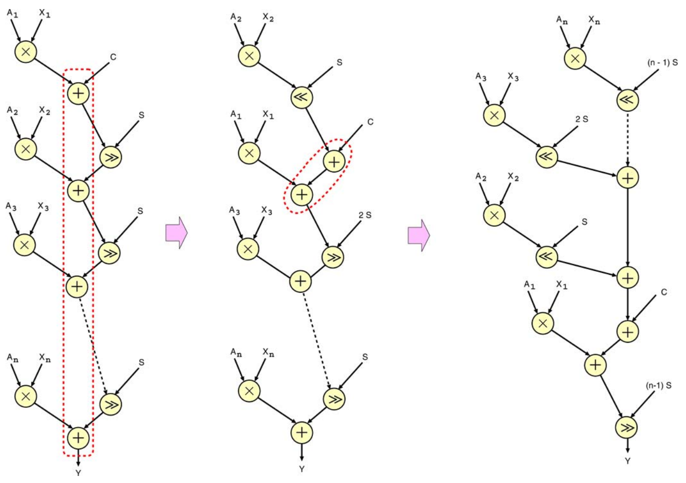


Fig. 30. Delay of an FIR filter kernel can be reduced significantly by applying our transformations which merge additions separated by right-shift operations.

V. Conclusion

We have presented an algorithm to optimize complex arithmetic circuits. We believe that too little work has been done in the area of arithmetic optimization: on one side, logic synthesis is unable to attain, alone, the potentially achievable results; on the other side, the importance of implementing automatically efficient arithmetic circuits is growing due to the proliferation of digital signal processing in ASIC and FPGA designs and to the need of implementing software-derived computational kernels in hardware accelerators, coprocessors, or special functional units.

Our algorithm integrates well into a traditional synthesis flow and prepares the dataflow graph for the logic synthesizer to make the best out of it. It is based on the extensive use of the carry-save representation and, compared to an existing work in the domain, exposes more optimization opportunities than previously available. On practical benchmarks, it improves the critical path by 20%–40% and reduces the area by 10%–20%. It is fast (runs for fractions of seconds on all our test cases), it generates a subset of all solutions possible within our problem formulation, and it guarantees that all Pareto-optimal solutions are in this subset.

References

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

[2] A. R. Omondi, *Computer Arithmetic Systems*. New York: Prentice-Hall, 1994.

[3] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Trans. Comput.*, vol. 47, no. 3, pp. 273–285, Mar 1998.

[4] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 294–306, Mar. 1996.

[5] G. De Micheli, *Synthesis and Optimization of Digital Circuit*. New York: McGraw-Hill, 1994.

[6] M. Potkonjak and J. Rabaey, "Maximally fast and arbitrarily fast implementation of linear computations," in *Proc. Int. Conf. Comput.-Aided Des.*, Santa Clara, CA, Nov. 1992, pp. 304–308.

[7] A. Peymandoust and G. De Micheli, "Symbolic algebra and timing driven data-flow synthesis," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2001, pp. 300–305.

[8] V. Dimitrov, L. Imbert, and A. Zakaluzny, "Multiplication by a constant is sublinear," in *Proc. 18th IEEE Symp. Comput. Arithmetic*, 2007, pp. 261–268.

[9] T. Kim, W. Jao, and S. Tjiang, "Circuit optimization using carry-save-adder cells," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 10, pp. 974–984, Oct. 1998.

[10] J. Um and T. Kim, "An optimal allocation of carry-save-adders in arithmetic circuits," *IEEE Trans. Comput.*, vol. 50, no. 3, pp. 215–333, Mar. 2001.

[11] Z. Yu, K.-Y. Khoo, and A. N. Willson, Jr., "Optimal joint module-selection and retiming with carry-save representation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 7, pp. 836–846, Jul. 2003.

[12] A. Mathur and S. Saluja, "Improved merging of datapath operators using information content and required precision analysis," in *Proc. 38th Des. Autom. Conf.*, Las Vegas, NV, Jun. 2001, pp. 462–467.

[13] *Creating High-Speed Data-Path Components—Application Note*, Synopsys, Mountain View, CA, Aug. 2001. version 2001.08.

[14] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2004, pp. 791–798.

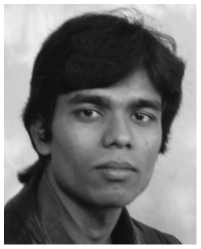
[15] V. G. Oklobdzija and D. Villeger, "Multiplier design utilizing improved column compression tree and optimized final adder in CMOS technology," in *Proc. Int. Symp. VLSI Technol., Syst. Appl.*, 1993, pp. 294–306.

[16] V. G. Oklobdzija and D. Villeger, "Improving multiplier design by using improved column compression tree and optimized final adder in CMOS technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, no. 2, pp. 292–301, Jun. 1995.

[17] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[18] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth cognizant architecture synthesis of custom hardware accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 11, pp. 1355–1371, Nov. 2001.

[19] S. Vassiliadis, E. A. Hakkennes, J. S. S. M. Wong, and G. G. Pechanek, "Sum-absolute-difference motion estimation accelerator," in *Proc. 24th Conf. EUROMICRO*, Vesteras, Sweden, Aug. 1998, pp. 20 559–20 566.



Ajay K. Verma received the B.S. degree in computer science from the Indian Institute of Technology Kanpur, in 2003. Since 2004, he has been working toward the Ph.D. degree in the Processor Architecture Laboratory, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

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 in 2007.

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

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

Dr. Brisk was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis 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 Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996.

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

Prof. 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, 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 has been a 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 (VLSI) Systems on Application Specific Processors. He was a recipient of the DAC 2003 and the CASES 2007 Best Paper Awards.