Improved Use of the Carry-Save Representation for the Synthesis of Complex Arithmetic Circuits

Ajay K. Verma
Processor Architecture Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
Lausanne, Switzerland
AjayKumar.Verma@epfl.ch

Paolo Ienne
Processor Architecture Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
Lausanne, Switzerland
Paolo.Ienne@epfl.ch

Abstract

The increasing importance of datapath circuits in complex systems-on-chip calls for special arithmetic optimisations. The goal is to achieve automatically the handcrafted results which escape classic logic optimisations. 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—especially 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 in 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%.

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, cryptography) are particularly resistant to known optimisation 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` [5]. The dataflow represents a slightly approximated 16×4 multiplier but its direct implementation, even applying known techniques described in literature, would result in a suboptimal implementation 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 [8]. Our goal is to achieve the speed of the latter implementation with automatic transformations of the dataflow graph.

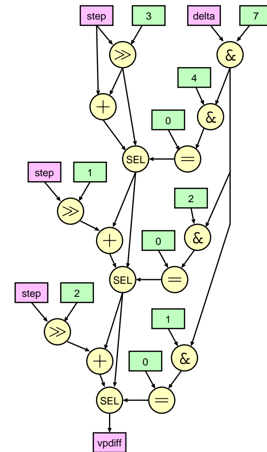

Figure 1. A typical dataflow graph obtained by parsing automatically the C-language bit-level description of a custom arithmetic operator.

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

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 [8]. 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 latest contributions, [11] discusses various heuristic and optimal algorithms to design arbitrary compressor trees (see Fig. 2). Our work focuses on exposing compressor trees in arithmetic computations and we use a simple heuristic similar to the *Three-Greedy Approach* [11] to demonstrate the strength of our technique in result section; our optimisation algorithm would simply profit by the availability of 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 [2]. Some works

have addressed the high-level arithmetic optimisation of important classes of computations (e.g., linear [10]) or, more recently, have used symbolic techniques to rewrite dataflow graphs to minimise their critical path [9]. 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. Kim *et al.* [4] discuss some simple transformations of the topology of the circuit to maximise the use of cascaded *Carry-Save Adders* (CSAs, see again Fig. 2)—which in fact constitute compressor trees. Yet, in both [4] and [13], the main emphasis is in the optimal allocation of the addenda to the cascaded CSAs—thus making these works somehow related to [11], albeit with a slightly different problem formulation (at bit-level in the latter vs. word-level in the former ones). Yu *et al.* [14] 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 [7], the authors 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 [12]; the opportunities seized by Synopsys's *Behavioural Optimisation of Arithmetic (BOA)* are probably similar to those described in [4]. Although somehow similar in the intents, our work is more radical 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 optimise some new cases of practical interest. This piece of work builds on a previous contribution of ours [3], where similar results were attained; yet, the algorithm presented here warrants some novel interesting properties and allows an exploration of Pareto-optimal design points.

ARITHMETIC OPTIMISATIONS

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

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

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 tree (which is feasible, as described in [11] or [13]).

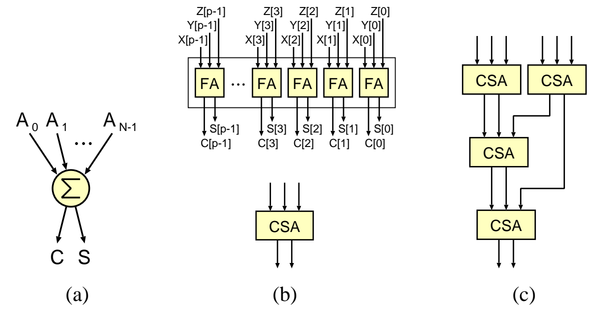

Figure 2. Compressor trees. (a) The symbol used here for a compressor tree. (b) A carry-save adder, which is the simplest 3-input compressor tree. (c) Carry-save adders used to build a compressor tree.

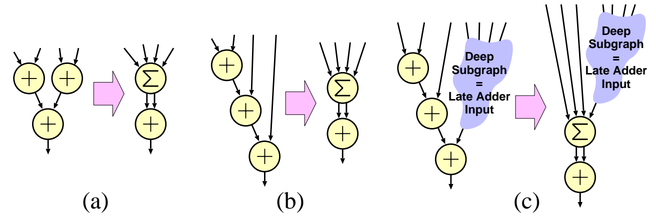

Figure 3. Example of application of compressor trees to reduce the critical path of arithmetic circuits. In the special case of (c), using a compressor tree might actually *increase* the circuit delay due to the late arrival of the last addendum.

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 situations and implicitly assume that the use of compressor trees wherever possible is beneficial. In fact, our compressor trees are built under the assumption of identical arrival time for all inputs. The assumption is valid if all the variables are coming from registers (which is true if we are optimising special functional units). In reality input arrival times may not be same and this is the main source of suboptimality—not the use of compressor trees in itself; a possible solution would therefore be to implement compressor trees with knowledge of the arrival times of the individual signals [11].

Combining Adders

We assume to 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

$$
\begin{array}{rcl}
-A & \Rightarrow & \overline{A}+1 \\
A-B & \Rightarrow & A+\overline{B}+1 \\
A*B & \Rightarrow & \mathsf{Sum}(\mathsf{PP}(A,B))
\end{array}
$$

Table 1. Rewriting rules for nonprimitive arithmetic operations.

it in hardware. Instead, to obtain faster and more efficient implementations, before writing out the graph for the synthesizer, we can apply three types of transformations to the graph with different goals:

Expose adders in other arithmetic operations. This is a rather classic set of transformations, just rewriting operations such as subtractions and multiplications as logic operations and additions. The transformations are listed in Table 1. The third transformation implements a parallel multiplier [8]: $\mathsf{PP}()$ produces a set of partial products and $\mathsf{Sum}()$ represents multiple-input additions which will be implemented in hardware using a compressor tree.

Rearrange adders to maximise multi-input adders. This is the fundamental set of rules to achieve our goal: both software constructs (e.g., if-conversion) and other logic operations (e.g., shifts, inversions, bitwise operations, etc. 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 optimisation will be missed. These rules are the main novelty of our approach and, conversely, their absence is the most severe limitation of other techniques discussed in the related work section. We first cluster adders by seperating them from other logic operations and then we merge them into multi-input adders.

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-propogate final adder.

Note that we do not implement any rule to optimise 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. In case 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 language. Although rare, this may happen in applications implementing modular arithmetic—namely, cryptography. We ignore such peculiarities.

Sorting Problem

We formalise 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 1. Nodes

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

Table 2. Existing possibilities for advancing class L operations over additions.

V represent primitive operations and edges E represent data dependencies. 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 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 (e.g., see Table 1). All other operators are logic nodes. The function $\mathrm{Class}(\cdot)$ 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 formalised 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* $\mathrm{Class}(u) = \mathsf{A}$ *and* $\mathrm{Class}(v) = \mathsf{L}$*, either it is always* $\mathrm{Ord}(u) < \mathrm{Ord}(v)$ *or always* $\mathrm{Ord}(u) > \mathrm{Ord}(v)$*.*

We call such a graph G' *sorted*. For instance, the motivational example of Fig. 1 is unsorted, whereas that of Fig. 12 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.

Sorting Rules

Table 2 summarises possible transformation rules to sort node couples and advance class L operations over addition. As it can be observed, only in some cases such swapping is possible. As shown in Fig. 4:

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

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

  Similarly left shifts (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.

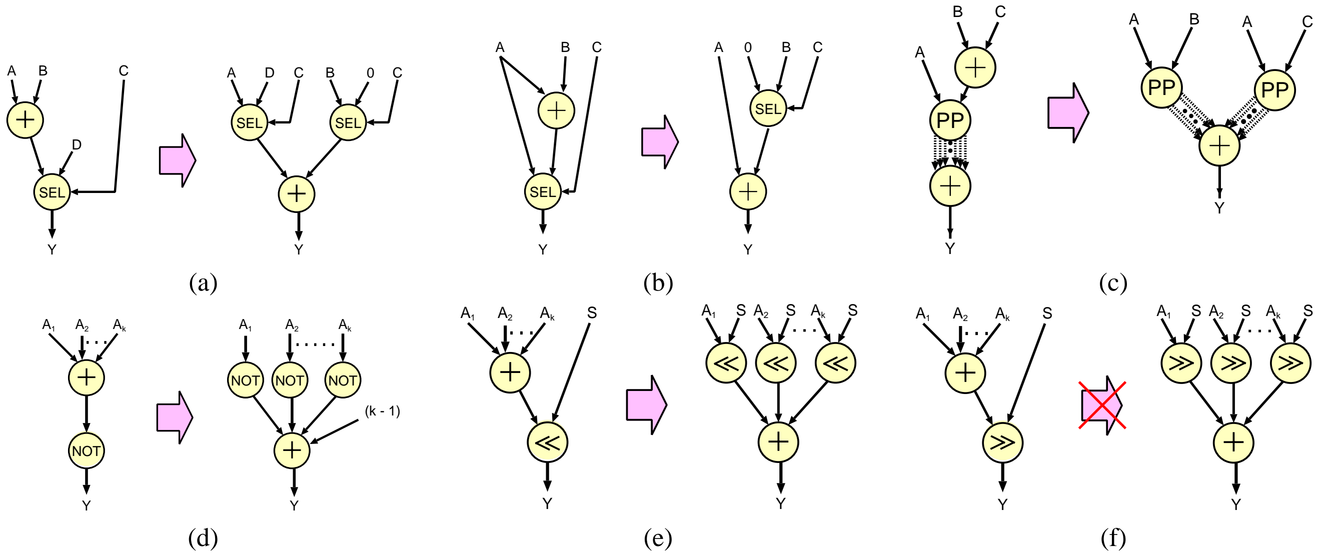

Figure 4. Advancing (a) selector over addition in the general case, (b) selector over addition when one of the input of addition is identical to the other input of the selector, (c) partial product over addition, (d) bitwise NOT over addition, and (e) shift left over addition. Note that (f) shift right can not be advanced over addition.

Formally we can write:

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

- A similar but less elementary transformation consists in advancing a selector node over addition; it is based on the existance 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, though, that if B is identical to one of the A_i (say 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.

- The last sorting rule consists in advancing PP over addition. This transformation is based on distributive property of multiplication over addition, but is significantly more complex than other transformations:

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

where $\mathcal{F}$ indicates the operators which, after prior transformations, 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 transformation is only applied when really beneficial.

We consider only sorting rules to advance logic operations over additions. No rules for advancing additions over logic nodes are known except in the case of a selector:

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

It can be easily shown that this transformation is redundant because selectors can also be advanced over addition; therefore, if this new transformation clusters two add nodes A_1 and A_2 by advancing A_2 over some selector, the same advantage can also be obtained by advancing such selector over A_1.

Issues with Sorting Rules

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: no sorting rules are available for basic bitwise logic operations, for effective right shifts, and for equality comparison (i.e., a bitwise XOR and an AND reduction operation). One could then apply any of the existing four sorting rules on the graph wherever possible to cluster additions as much as possible. The question on what is the most appropriate order in which to apply the sorting rules naturally arises.

Order Independence

In fact, the sorting rules constitute a finitely terminating and confluent (i.e., convergent) reduction system [1], and thus the order is immaterial to the result.

Property 1 *Given a dataflow graph G, any possible sequence of transformations resulting in a graph G' which cannot be further transformed results in the same final graph G'.*

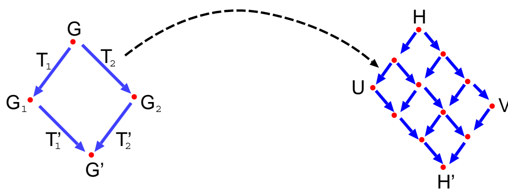

Figure 5. Local confluence and the other properties imply convergence of the sorting rules.

The proof depends on the following properties:

- **Topology independence.** All sorting rules are independent of the topology of the graph—e.g., we can always advance left shifts over additions, no matter the topological structure of the graph.
- **Persistency.** All four sorting rules are such that they preserve all other transformations. That is, if there is a transformation T_2 applicable before applying transformation T_1, then it is also applicable after applying T_1. This can be easily verified by considering all 16 cases (4 possibilities for both T_1 and T_2).
- **Local confluence.** Given a graph G, one can prove that if, after applying sorting rules T_1 and T_2, one gets graphs G_1 and G_2, then there are 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. The proof of this property is again based on considering all 16 cases.

The proof essentially uses recursively local confluence to prove global confluence (Fig. 5). The possible graphs obtained with the sorting rules forms a lattice.

Useful Mobility

The next issue is that actually our transformations do not necessarily reduce the critical path delay: except for the fourth transformation (advancing PP over addition), all other transformations reduce the critical path exclusively if they result in merging two addition nodes. Some of these transformations 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. So one would like to use these transformations 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 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 which are susceptible to bring an advantage if a sorting rule is applied to them. We mark movable nodes as useful if any of the following cases holds true:

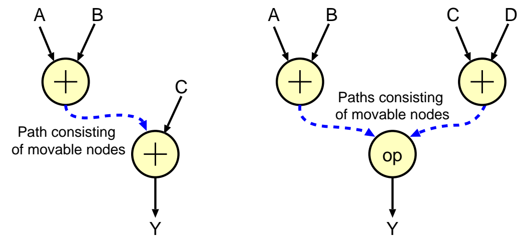

Figure 6. All the nodes in the path are useful mobile nodes.

- They lay on a path between two additions, and such path consists exclusively of movable nodes (Fig. 6).
- They lay on one of two paths connecting two addition nodes with a single movable node, and both paths consists exclusively of movable nodes. The joining movable node is also a useful movable node (Fig. 6).

(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 above 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.

Impractical Transformations

The last issue is that there might be some transformations which will result in merging addition nodes but may have a large area cost associated with them. Such transformations 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.

Area-timing Design Space Exploration

We want to find all Pareto-optimal graphs which can be obtained from a given graph by applying the sorting rules described in sorting rules section. Fig. 7 shows an algorithm to generate all solution. The algorithm is conservative in that it does not guarantee that all generated solutions are Pareto-optimal.

The algorithm consists of three steps: (1) In the first step we rewrite operations according to the rules of Table 1 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. (3) In the third step, which is the crucial one, we generate all Pareto graphs. Fig. 8 shows this last step. Essentially, we split the sorting rules into two classes: those which increase area and those which do not. The idea is to apply each class of rules independently: while area-constant rules are applicable, no Pareto-optimal solution can appear.

```
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(ℒ, Graph) {
  Graph g;
  g = introduceCompressor(Graph);
  ℒ.append(g); }

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

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

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

  markUsefulMobile(Graph);

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

Figure 7. The optimisation algorithm.

Once area-constant rules are not applicable anymore, solutions are potentially Pareto-optimal ($G_1, G_2, ..$ in the Fig. 8). Then, all area-increasing rules can be applied and the process reiterated until the final graph G' is reached. In this way, we can generate all graph susceptible of being Pareto optimal without generating all solutions.

One can see that in the function *applyCostlessTransformations()* we are also doing some clean-up transformations: we propagate constants, introduced in some of the transformations (such as advancing bitwise NOT over addition), we merge adjacent left shifts, and we merge some specific instances of selector nodes as shown in Figure 9. At this stage, we also perform bitwidth analysis for the reasons outlined in the next section.

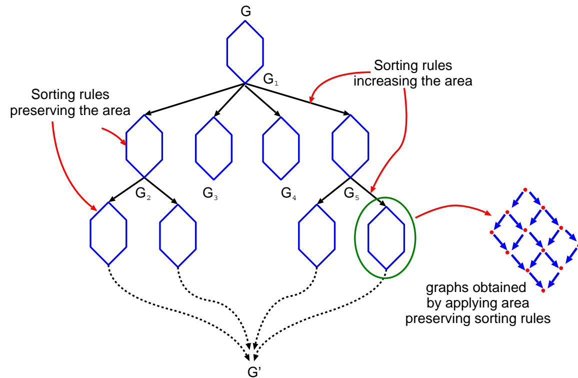

Figure 8. Finding all Pareto solutions without generating all graphs.

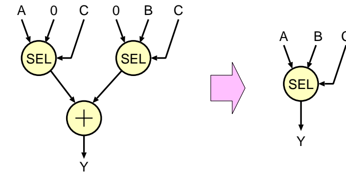

Figure 9. Merging two selector nodes into a single selector.

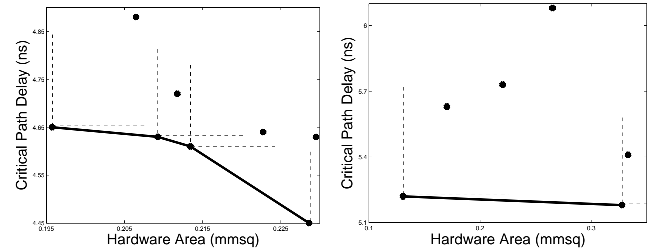

Figure 10. Solutions reported by our algorithm for the Video Mixer (left) and Polynomial of 4th Degree (optimised, right).

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, 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 optimisations 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 analyse the number of bits required for every variable in the dataflow graph. To obtain this information, we have implemented a bitwidth analysis, very similar to the algorithms described in [6] and probably similar to the analysis that the synthesizer we use implements internally. We use this bitwidth analysis pass as follows: During optimisation, we assume full precision everywhere in the graph (32 bits in our case) and apply all rewriting rules (notably the multiplication rewriting rule, Table 1) accordingly. Then, once the optimisation algorithm described in the previous section has been run, we apply the simplification loop shown in Fig. 11 to prune the graph from unnecessary edges. Multiple iterations are needed until convergence is attained, because the reduction of input edges

Benchmark	Original		Synopsys BOA [12]		**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 [5]	1.92	.015	1.92 (0%)	.015	1.04 **(-46%)**	.013 **(-14%)**	1.07 **(-44%)**	.012 **(-23%)**
manually optimised	1.18	.010						
G.721 [5]	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 4th Degree (optimised)	5.22	.130	5.64 (+8%)	.144	5.18 **(-0.8%)**	.328 **(+151%)**	5.22 **(0%)**	.131 **(0%)**
Polynomial 4th Degree (Horner)	6.67	.096	6.43 (-4%)	.104	5.65 **(-15%)**	.887 **(+833%)**	6.67 **(0%)**	.096 **(0%)**
Video Mixer [12]	4.88	.206	4.46 (-9%)	.143	4.45 **(-9%)**	.229 **(+14%)**	4.65 **(-5%)**	.196 **(-6%)**

Table 3. Results of our arithmetic optimisations on some benchmarks.

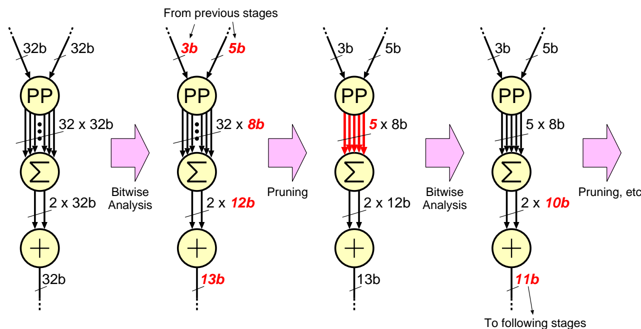

Figure 11. An example of application of the bitwidth analysis to prune a graph.

in compressors may reduce the precision of their outputs. These in turn trigger further pruning later in the graph.

EXPERIMENTAL RESULTS

We have written an experimental optimiser 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 optimised graphs are converted to VHDL and synthesized with a recent version of Synopsys's *Design Compiler* with the arithmetic generators of the *DesignWare* library. For the optimised 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 [11].

We have run several benchmarks through our optimiser pass and we have synthesized the results for an ASIC library of a common $0.18\mu m$ CMOS technology. The results are shown in Table 3. The "Original" columns refer to the direct synthesis of the dataflow graph. "Synopsys BOA" uses the optimisations available in some versions of Synopsys's *Design Compiler* and in *Behavioral Compiler* as mentioned in related work section . The results of the columns "Our Arithmetic Optimisations" are obtained with the algorithm of Fig. 7. Here also we have two types of columns: the first columns show the Pareto solution with minumum 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 optimisations; 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 3 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 [5]: 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 dataflow graph does not contain SEL nodes. The bottom four 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, and the last is a video mixer application appropriate for demonstrating classic arithmetic optimisations [12].

In Table 3, reading the results from the bottom group, one can see that on purely arithmetic benchmarks our optimisation techniques achieve practically the same results of existing optimisers. In the cases of the first group, where logic operations are scattered among arithmetic computations, BOA misses completely the potentials for optimisation as would the other algorithms described in 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 optimised 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%.

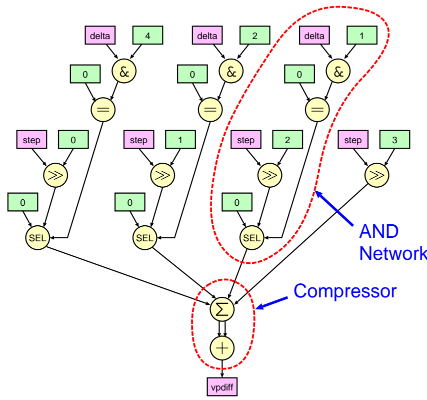

Figure 12. The motivational example of Figure 1 after optimisation is fully sorted. The implementation is optimal, as suggested also by Table 3 (ADPCM Decoder).

For two benchmarks we plotted on a delay-area graph the few potential Pareto-optimal solutions reported by our algorithm (Fig. 10). Notice that all Pareto-optimal solutions 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 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. In previous work, we only generated a heuristic solution close to the fastest [3].
As an example, Fig. 12 shows the fastest solution corresponding to the graph of 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 simple set of AND gates.

CONCLUSIONS

We have presented an algorithm to optimise complex arithmetic circuits. Apart from a few pieces of previous work, we believe that too little work has been done in the area of arithmetic optimisation: 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 simply prepares the dataflow graph for the logic synthesizer to make the best out of it. It is based on extensive use of the carry-save representation and, compared to existing work in the domain, exposes more optimisation opportunities that previously available. On practical benchmarks, it improves the critical path of 20–40% and reduces the area by 10–20%. It is fast (runs for fractions of seconds on all our testcases) and generates a subset of all solutions possible within our problem formulation, and guarantees that all Pareto-optimal solutions are in this subset.

REFERENCES

[1] Baader, F., and Nipkow, T. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.

[2] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[3] Ienne, P., and Verma, A. K. Arithmetic transformations to maximise the use of compressor trees. In *Proceedings of the IEEE International Workshop on Electronic Design, Test and Applications (DELTA)* (Perth, Australia, Jan. 2004).

[4] Kim, T., Jao, W., and Tjiang, S. Circuit optimization using carry-save-adder cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-17*, 10 (Oct. 1998), 974–84.

[5] Lee, C., Potkonjak, M., and Mangione-Smith, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (Research Triangle Park, N.C., Dec. 1997), pp. 330–35.

[6] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-20*, 11 (Nov. 2001), 1355–71.

[7] Mathur, A., and Saluja, S. Improved merging of datapath operators using information content and required precision analysis. In *Proceedings of the 38th Design Automation Conference* (Las Vegas, Nev., June 2001), pp. 462–67.

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

[9] Peymandoust, A., and De Micheli, G. Symbolic algebra and timing driven data-flow synthesis. In *Proceedings of the International Conference on Computer Aided Design* (San Jose, Calif., Nov. 2001), pp. 300–5.

[10] Potkonjak, M., and Rabaey, J. Maximally fast and arbitrarily fast implementation of linear computations. In *Proceedings of the International Conference on Computer Aided Design* (Santa Clara, Calif., Nov. 1992), pp. 304–8.

[11] Stelling, P. F., Martel, C. U., Oklobdzija, V. G., and Ravi, R. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers C-47*, 3 (Mar. 1998), 273–285.

[12] Synopsys. *Creating High-Speed Data-Path Components—Application Note*, Aug. 2001. Version 2001.08.

[13] Um, J., and Kim, T. An optimal allocation of carry-save-adders in arithmetic circuits. *IEEE Transactions on Computers C-50*, 3 (Mar. 2001), 215–233.

[14] Yu, Z., Khoo, K.-Y., and Willson, Jr., A. N. Optimal joint module-selection and retiming with carry-save representation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-22*, 7 (July 2003), 836–46.