Towards the Automatic Exploration of Arithmetic-Circuit Architectures

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

ABSTRACT

The optimization of arithmetic circuits has always been essentially a manual task: arithmetic experts study the best architectures for arithmetic components and write libraries of generators, and designers instantiate library components and rely on logic synthesizers to obtain good implementations. In this paper we look at the capabilities of commercial synthesizers when it comes to arithmetic circuits, and observe that they are essentially unable to switch from one arithmetic architecture to another (e.g., from a ripple-carry to a carry-lookahead adder). Therefore, users relying on logic synthesis miss most optimization potentials. We therefore investigate algorithms for factorization which can *prepare* structured VHDL or Verilog for synthesizers to implement, and show first steps into pruning the search space from many irrelevant or equivalent solutions. Our results are still very limited in complexity but we show that our techniques successfully concentrate on the automatic exploration of very different solutions, and *discover* architectures known and unknown to expert designers, such as different types of adders, the carry-save representation, or improved multipliers. This is a first step toward a class of arithmetic optimizers which sit on top of classic logic synthesizers.

1. INTRODUCTION

The exponential increase in complexity of *Application-Specific Integrated Circuits (ASICs)* and the ever growing requirements in terms of performance and energy efficiency demand more pervasively complex digital arithmetic circuits finely tuned to an application. Such circuits for signal and media processing are typically at the heart of dedicated hardware accelerators connected to processors or are customized parts of RISC or VLIW processors where they implement rich instruction-set extensions. Although arithmetic circuits are still, in many cases, hand crafted by experienced designers, the reduced time available for a design cycle make it increasingly desirable to run arithmetic designs through the standard design flow of synthesis with standard cells and timing-driven place and route.

When this is the case, libraries of standard arithmetic components in the form of parametric generators are often used in the front-end of logic synthesizers—DesignWare by Synopsys is a well-known and widely used example. The role of logic synthesis in the design of complex arithmetic circuits is relatively minor: arithmetic circuits usually involve extremely large combinatorial blocks which can only be locally optimized through the capabilities of typical synthesizers. This limitation is circumvented to some extent by supplying logic synthesizers with partially structured arithmetic components: arithmetic generators implement specific architectures (e.g., adders in carry-propagate or carry-lookahead form, depending on the high-level optimization goals) often in the form of automatically-generated technology independent netlists whose handcrafted structure guides logic synthesizers in the right direction. This approach has the drawback of still requiring experts to prepare the libraries in the first place, and, more significantly, misses any serious optimization across individual arithmetic components. Some synthesizers, such as Design Compiler Ultra, make up for this lack with limited ad-hoc optimizations—e.g., to infer the use of the carry-save representation [10]. Besides these, the burden stays with the designers.

In this paper we make an attempt to understand what it takes to automatically synthesize optimal arithmetic structures, in complete independence from specific arithmetic architectures known *a priori*. Essentially, this turns out to be an instance of the well-known problem of *multiple-level logic optimization* [5] where we aggressively try to avoid heuristic simplifications. We make first steps in simplifying the problem in such a way as to exploit at best the capabilities of logic synthesizers and concentrate the architectural exploration on the part that these cannot achieve. Our final goal would be to add an arithmetic-level optimizer in front of any logic synthesizer which would automatically structure any arithmetic circuit much in the same way as handcrafted components from traditional libraries are. We exemplify what we mean by *structuring* in the next section.

2. MOTIVATIONAL EXAMPLE

An adder can be described in many ways, as symbolically shown in Fig. 1. The first one is what we describe as a *flat* adder in which all the output bits are calculated indepen-

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

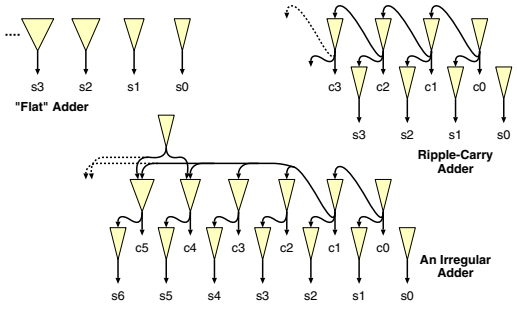

Figure 1: Three adders with different delay and area: the different forms of reuse of common subexpressions across different bits of the output heavily influence their performances.

dently. The second one is a typical ripple-carry adder, where for the computation of each sum and carry bit, the previous carry is used. The third one is an adder where, in the computation of some of the carry bits, previous carry bits are used, while other carry bits are computed independently.

Although the three circuits represent the same function, a synthesizer gives significantly different results for the three adders: in a given technology, synthesizing with maximum optimization effort for best delay, the flat adder results in a delay of 0.49ns and an area of $691\mu m^2$, the ripple-carry adder of 0.41ns and $385\mu m^2$, and the third one of 0.34ns and $534\mu m^2$. At this point one might think that flat adder is the one which should have minimum delay, but since the currently used heuristics are not able to factorize the flat expression optimally, it results in a circuit with significantly larger delay and area as compare to other circuits. Ideally, if one gives any description of an adder to the synthesizer and asks for an implementation with best area, one should automatically get the ripple-carry adder or a similar design, while if one asks for an implementation with smallest delay, the irregular adder should be implemented. Yet, synthesizers optimize circuits only somehow locally and seldom cross "architectural boundaries" between different classes of implementations of arithmetic components. Our goal is to supply synthesizers with structured descriptions which make them synthesize the best architecture in every instance.

What is different between these three adders? Not the logic functions, because of course they are all producing the same outputs. There are two main differences: The first difference is in the *Common Sub-Expressions (CSEs)* expressing the reuse of prior intermediate results for further computations (e.g., use the carry of the previous bit as a starting point to compute a given sum bit), and, the second difference is in the in the factorization of the expressions. Clearly, this is the additional information which the synthesizer needs and exploits; if such information is missing, the synthesizer uses its own heuristics to factorize the expressions and find CSEs, yielding suboptimal results. What we need to build is a preprocessor rewriting the logic functions of a complex arithmetic component in all possible ways and thus using all combinations of CSEs.

3. STATE OF THE ART

Computer arithmetic has been for decades one of the richest areas of research in design optimization, mostly by handcrafting ever improving architectures for all fundamental circuits of interest [7]. Only rare cases of algorithmic optimizations for very specific circuits have been reported in the literature; examples include variable group-size carry lookahead adders [6] and irregular partial-product compressors for multipliers [9]. These examples correspond to typical manual optimizations which have been formulated as solvable optimization problems but are, of course, absolutely specific to particular components and architectures. More general approaches still address only very specific classes of optimizations; an example is the inference of carry-save structures in some peculiar situations [11].

The general problems of multiple-level logic optimization and of factorization has been usually addressed without special attention to arithmetic circuits. It is an old well-studied problem [5] and it is fundamentally related to division. Classic work on *kernels* (such as [2, 3]) is based on algebraic division, which is particularly ineffective in XOR dominated circuits, as most arithmetic circuits are. Emphasis in the late eighties and early nineties was often on heuristics to exploit *don't care* situations—an important case in control logic and finite-state machines, but almost inexistent in arithmetic circuits. On the other hand, we focus here on the optimization of AND-XOR forms, and on the search of all Pareto-optimal circuits rather than on individual heuristic solutions.

Boolean division is known to be a superior way of finding common terms [5, 3], especially useful in XOR-rich circuits, but it is very expensive computationally despite efforts to make it more efficient [4]. We will use a division methodology close to the Boolean one but based on Gröbner bases [1], as used in commercial symbolic algebra packages, and on describing the circuit to implement in the Reed-Muller form [3]. Symbolic algebra has received considerable attention in the recent years as a way to simplify complex computation in hardware, in software, and to improve the reuse of special instructions (e.g., see [8] for one of the earliest works in the area). The work of Peymandoust and De Micheli is focused at the word level—that is, to try to make the best use of components from existing arithmetic libraries. We will attempt, in a not dissimilar way, to use symbolic algebra but at the bit level—that is, to implement better such components.

4. PROBLEM DEFINITION

Given a set of Boolean functions representing an arithmetic component, our goal is to enumerate all possible factorizations which expose CSEs common to more than one of the functions in the set. In particular, we look for all those which (1) are Pareto-optimal—that is, circuits that are better in area and/or in critical-path delay than any other—and which (2) are not equivalent for logic synthesis.

We aim a tool which takes any description of the functions to implement (such as the VHDL corresponding to the Flat Adder of Fig. 1) and produces synthesizable descriptions which have different CSEs exposed between the various functions (such as the VHDL corresponding to the other adders of Fig. 1). We want to leverage the strength of common logic synthesizers in optimizing simple expressions but we want to help the synthesizer by pre-structuring appropriately the circuit. For this, we consider *equivalent for logic synthesis* (although this might not be perfectly true in every case) any factorization that differs only in the way a subexpression is factorized. In other words, two descrip-

tions are not equivalent for us if at least one of the CSEs shared by two or more functions in one of the descriptions is different from all CSEs in the other description. Limiting ourselves to solutions which are not equivalent in this sense will help us reduce considerably the search space.

Additionally, we also reduce the search space by not pursuing particular solutions as soon as we can show that they cannot be Pareto-optimal. We build an implicit enumeration tree exploring all possible CSEs and our problem is essentially that of pruning it efficiently from all the leaves and branches that are either guaranteed to be Pareto-inferior or equivalent for logic synthesis. We call this problem *selective factorization* and describe our first attempt to its solution in the Section 6. In the next section we discuss the fundamental building block of our factorization mechanism.

5. GRÖBNER BASES AND DIVISION

The problem of factorization is directly related to division. Algebraic division is a fast way of performing division but it misses many results of practical interest; Boolean division gives much better results, but it is computationally very intensive. One of the problems in algebraic division is that it treats literals such as a and $\bar{a}$ independently and hence ignores Boolean properties like $a \,\mathsf{AND}\, \bar{a} = 0$ or $a \,\mathsf{OR}\, \bar{a} = 1$. For instance, let's consider the two expressions $E_1 = a+b+c$ and $E_2 = a \cdot b \cdot c \;\mathsf{OR}\; \bar{a} \cdot \bar{b} \cdot c \;\mathsf{OR}\; a \cdot \bar{b} \cdot \bar{c} \;\mathsf{OR}\; \bar{a} \cdot b \cdot \bar{c}$, where $+$ indicates XOR and $\cdot$ indicates AND. It is easy to see that two expressions are identical and hence the common expression of E_1 and E_2 should be the whole expression; algebraic division will fail to recognise this fact.

Gröbner bases are a well known method used for multinomial division. It is based on the fact that polynomials over a field ($\mathbb{R}[x_1, x_2, \cdots, x_m]$) form a ring. Since Boolean expression with operations AND and OR do not form a ring, we cannot use Gröbner division for Boolean expressions. However if we convert the Boolean expressions into Reed-Muller form (XOR-sum of product form [3]), we obtain a ring over the field $GF(2)$ and we can apply Gröbner bases. We use the function `reduce` in Maple 9 for multinomial division. This function takes two expressions—dividend (A) and divisor (B)—and an ordering among the variables as inputs and divides A by B in such a way that the resulting remainder is lexicographically smallest according to the input ordering among variables. We use the AND or the XOR of a pair of variables as divisor and try all possible ordering among the variables. Note that Maple does not recognise the Boolean property $x^2 = x$. We force this property in one direction by dividing the remainder by $(x^2 - x)$, which will replace all occurrences of x^2 by x; on the other hand, we cannot do the opposite transformation, and therefore the division we use is less effective than Boolean division.

6. SELECTIVE FACTORIZATION

In this section we discuss how we search for possible factorizations with CSEs, and prune the space of interesting factorizations while we proceed. Some pruning of the infinitely many implementations of a circuit is absolutely trivial: any circuit can be transformed in one equivalent by inserting a couple of inverters on a wire, and the resulting circuit is clearly not Pareto-optimal. Unfortunately, avoiding these solutions is by far not sufficient to contain the number of useful solutions and we use a number of techniques

```
getPareto (function f = (E1, E2)) {
  Set S = φ;
  Set F = φ;
  forall common subexpressions S between E1 and E2 {
    S = getPareto(S);
    E1' = replace(S, s, E1);
    E2' = replace(S, s, E2);
    // E1' and E2' are the expressions which are
    // obtained by replacing subexpression S in E1
    // and E2 by a variable s.
    Set E1' = getPareto(E1');
    Set E2' = getPareto(E2');
    forall Sk ∈ S {
      Set E1 = φ;
      Set E2 = φ;
      Set D = φ;
      forall E ∈ E1' {
        insert(replace(s, Sk, E), E1);
        insert(delay(replace(s, Sk, E)), D) }
      forall E ∈ E2' {
        insert(replace(s, Sk, E), E2);
        insert(delay(replace(s, Ss, E)), D) }
      forall D ∈ D {
        E1i = findMin(E1, D);
        E2j = findMin(E2, D);
        // findMin(S, D) finds the implementation
        // from the set S whose delay is bounded by
        // D, and has the smallest area among all
        // such implementations.
        insert((E1i, E2j), F); } } }
  return F; }
```

Figure 2: Algorithm to compute the Pareto-optimal implementations of two functions E_1 and E_2.

to further reduce them. Note that mostly our techniques are conservative—i.e., they never discard a Pareto-optimal implementation but conversely also retain nonoptimal ones.

We begin in the next section by splitting our problem into two: (1) enumerating all possible CSEs across several logic functions and (2) identify all Pareto-optimal implementations of a single function. We then consider in Sections 6.2 and 6.3 various techniques to solve the first problem efficiently. We conclude in Section 6.4 by showing that the second problem is nothing else than a special case of the general problem, and suggest how it can be handled.

6.1 Two Independent Problems

Two theorems help us split the problem at hand into two independent ones. The first theorem essentially states that when we look at a CSE across two or more functions, we are interested only in Pareto-optimal implementations of the CSE to obtain Pareto-optimal implementations of the functions. Now a natural approach to compute all Pareto-optimal implementations of E_1 and E_2 is to find all the CSEs and their Pareto-optimal implementations. For each implementation S_k of a CSE, we should compute all implementations of E_1 and E_2 which use S_k as the implementation of the corresponding CSE. If there are m such implementations of E_1 and n such implementations of E_2, then, by considering all the $m \cdot n$ combinations, we can get all Pareto-optimal implementations of E_1 and E_2. However thanks to a second theorem whose proof we omit, we do not need to consider all $m \cdot n$ combinations: instead, only $m + n$ are sufficient to generate all Pareto-optimal implementations.

Fig. 2 shows the algorithm based on the two theorems that generates all Pareto-optimal implementations of E_1 and E_2. The algorithm can be easily extended to more than two functions. We have thus partitioned the problem of finding all Pareto-optimal implementations of a set of functions

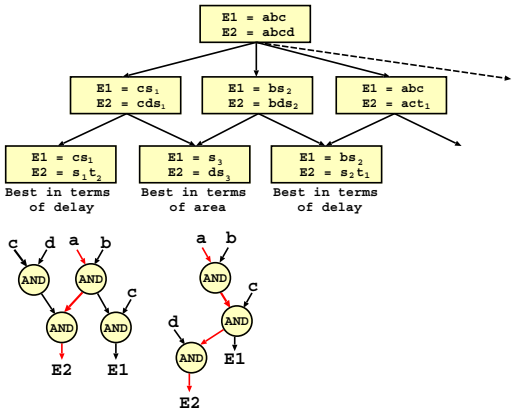

Figure 3: CSE-enumeration between two functions. Considering the maximal CSE, one might actually increase the critical path delay.

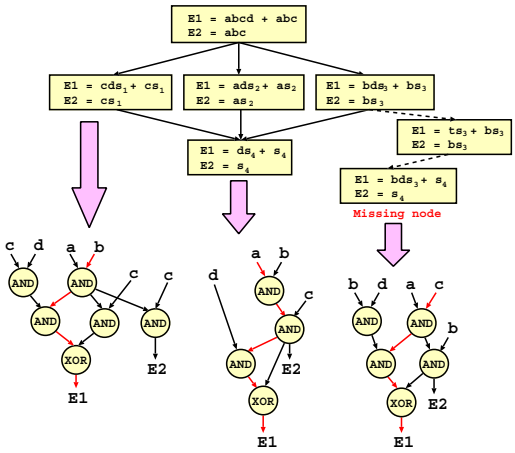

Figure 4: Using only s-reductions, one might miss Pareto-optimal implementations.

into two independent problems: The first problem consists in finding all CSEs between each pair of functions. The second problem consists of finding all Pareto-optimal implementations of a single expression. We will consider the two problems separately in the coming sections.

6.2 Basic Strategy to Find All CSEs

For the sake of simplicity we will consider only CSEs across two functions at a time; the generalization to multiple functions is trivial. To enumerate all CSEs, we build a *Direct Acyclic Graph (DAG)* where each node corresponds to a partial implementation of the two expressions with some sharing between them (an example is shown in Fig. 3). The node at the root of the DAG corresponds to the original expressions of the two output bits and the leaves of the DAG correspond to full implementations of the two expressions. At each node of the DAG we define some reductions to other nodes. A reduction takes two variables a and b from one of the two expressions and either replaces all occurrences of $a + b$ by a new variable, or replaces all occurrences of $a \cdot b$ by a new variable. Note that sometimes even partial occurrences of $a + b$ can be replaced by new variables (e.g.,

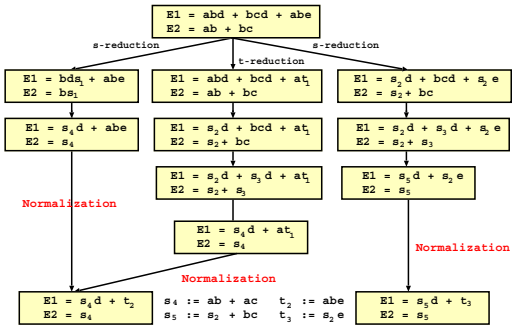

Figure 5: Effective values of s-variables and the canonical form of a node.

$a = (a + b) + b = s + b$), and such replacements can also be useful in reducing hardware area. We class these new variables in two types: *s-variables* if they occur in both expressions (and we call the corresponding reduction an *s-reduction*), or *t-variables*, if they occur only in one of the two expressions (the reduction is then a *t-reduction*). The DAG can be constructed in a depth-first manner using a recursive algorithm. All the nodes except the ones which are derived by t-reductions correspond to a shared subexpression within a single expression. The s-variables correspond to the CSEs we are interested into.

One might think that, since we are interested in finding the CSEs of two expressions, we should consider only s-reductions. Yet, considering only s-reductions we may miss some possible CSEs between two expressions which might indeed lead to a Pareto-optimal implementation. An example of this situation is illustrated in Fig. 4, where the missing node corresponds to a Pareto-optimal implementation.

However, most of the time the t-reductions are unnecessary and can be avoided. Note that at any node in the DAG there are $O(k^2)$ transformations possible where k is the number of variables in the Boolean expressions corresponding to the current node. Since in each transformation the amortised reduction in the sizes of Boolean expression is $O(1)$, the number of transformations from the root to the leave nodes can be as large as the size of the original Boolean expression. Hence, without any pruning, the size of the DAG can be up to $O((n+m)^{2m})$, where n is the number of variables in the original Boolean expressions and m is the size of the original Boolean expressions. In the next section we discuss the techniques to reduce the growth of the DAG.

6.3 Pruning the CSE Enumeration Graph

A given expression can be written in various ways, and therefore the same variable s can be obtained on many paths in the DAG. For instance, the variable $s = a \cdot b + a \cdot c$ can be obtained in two ways: either $s_1 = b + c$, $s = a \cdot s_1$, or $s_2 = a \cdot b$, $s_3 = a \cdot c$, and $s = s_2 + s_3$. Hence, we need to recognise identical CSEs because they may correspond to equivalent nodes in the enumeration DAG. On the other hand, one should notice that if a subexpression s_1 of s is used not only to compute one of the copies of s but also elsewhere for a different purpose, the two copies of s should not be considered equivalent as they can grant different future sharing potentials descending in the DAG. To check the equivalence of two s-variables we define the *effective value* of an s-variable: it represents the value of s expressed in terms

```
checkNeutral (function f = (E1, E2), var a, var b) {
  if(!isTReduction((E1, E2), a·b)) return false;
  expression E, abCoeff, aCoeff, bCoeff;
  if(!isPresent(E1, a·b)) E = E1;
  else E = E2;
  abCoeff = coeff(E, a·b);
  aCoeff = coeff(E, a) - abCoeff;
  bCoeff = coeff(E, b) - abCoeff;
  forall variables x in abCoeff {
    if(!isTReduction((E1, E2), a·x)) return false;
    if(!isTReduction((E1, E2), b·x)) return false; }
  if(1 ∈ terms(abCoeff)) {
    forall terms t in bCoeff {
      if(!isTReduction((E1, E2), a+t)) return false; }
    forall terms t in aCoeff {
      if(!isTReduction((E1, E2), b+t)) return false; } }
  return true; }
```

Figure 6: Algorithm for checking whether a reduction $t = a \cdot b$ is neutral.

of all the s-variables which are used also elsewhere and of primary inputs. The same effective value of two s variables indicates that they can be treated identically—e.g., unified by renaming them with an identical name. The effective value of s_i can be easily obtained by recursively substituting the effective values of only the s-variables which are used exclusively in s_i. A system of hash tables is used to efficiently check whether an s-variable with a given effective value exists. A similar equivalence can be established by defining in an analogous way the effective value of t-variables. This strategy helps in reducing the DAG size considerably because by unifying the s- and t-variables, different nodes may become the same. Examples of effective values of s_i's and t_i's are shown in Fig. 5.

One can notice that even after merging the DAG nodes using the above strategy there might exist two different nodes which have the same CSEs but which are treated differently. This happens due to their different structure in terms of t's and it is possible that one formulation may later lead to a sharing between the two output expressions which cannot be obtained through the other one. We define two nodes as *effectively equivalent* if the latter possibility can be ruled out. In principle, we should identify all effectively equivalent nodes and merge them into a single node; unfortunately, the problem of deciding whether two nodes are effectively equivalent is nontrivial. In order to check the equivalence of two nodes we define a canonical form, which is obtained by applying on the expression of a node all the applicable t-reductions which are not preventing any s-reduction (we call such reductions *neutral t-reductions*). At every node, we replace its expression with its canonical form. In addition to helping in checking for equivalent nodes, this also reduces the number of branches corresponding to neutral t-reductions. The algorithm for checking whether, for two variables a and b, the reduction $t = a \cdot b$ is neutral is shown in Fig. 6. The algorithm enlists all the reductions which are killed by this reduction and checks if any of them is a s-reduction. In case of algebraic division this would be an exact algorithm; however we might actually omit in rare cases some s-reductions (no useful s-reduction was ever prevented in our experiments, though). This is a case where our pruning of the enumeration DAG is not strictly conservative. At this point we do not check for $t = a + b$ being a neutral reduction because of the higher computation cost and of the more significant risk to to prevent useful s-reductions.

The last technique that we use is another heuristic which works rather effectively and did not prevent any further s-reductions in any of the cases considered. This is based on delaying a t-reduction as much as possible and hence reducing the number of branches emanating from a node. Suppose that at some particular node there exists a s-reduction which does not increase the minimum delay of the function; then, at that node we need not to apply any t-reductions. The reason is that t-reductions serve to prevent increases in the critical path delay in the corresponding branch, and that purpose is now served by the s-reduction. The problem with this approach is that we do not know how to calculate the minimum delay of a circuit because this in turn requires factorization. However, in some specific cases, it is possible to compare the minimum delay of two circuits without any additional assumption.

6.4 Pareto-Optimal Implementations of a Single Expression

The same approach used for multiple functions can be used with minor modifications to find all Pareto-optimal implementations of a single expression, and hence this becomes a special case of the global problem. In this case too, we define s-reductions, but here this indicates that the corresponding subexpression is used more than once in the expression; otherwise it is a t-reduction. If there are no s-reductions applicable, the expression has only a single Pareto-optimal implementation, which can be obtained by applying a two-greedy approach.

7. EXPERIMENTS

We implemented our algorithm using Maple 9 and we used it as a front-end of Synopsys Design Compiler. The input to our tool is the description of the flat functionality to implement; in principle, the input could be any unoptimized VHDL description of the arithmetic component. We output VHDL for all results and we synthesize them. Of course, we envisage the use of some rough estimators to select the small group of design that we actually want to synthesize—at the current point in time we simply consider all solutions.

We have implemented three qualitatively different circuits: adders, a parallel multiplier, and a three-input adder. Our goal is to show that our method explores all architectures of interest. Essentially, these should include the ripple-carry adder (CPA) and carry-lookahead adder (CLA) for the addition, the combination of a carry-save compressor and final adder for the multiplier, and the use of carry-save addition (CSA) for the three-input adder.

Table 1 shows the results of the optimizations. "Flat" indicates the input of our tool if synthesized directly. "Best Area" and "Best Delay" are two of the designs resulting from our optimizations (the best area indicated for the multiplier is only the best area between a subset of the results which we actually synthesized). The other entries represent either manual structural designs ("CPA") or the synthesis of components from DesignWare ("CLA" and "DesignWare"). As expected, our algorithm finds all optimal circuits.

Fig. 7 shows the area and delay of all 6-bit adders generated by our algorithm. One can notice that there are a handful of Pareto-optimal solutions (with only a limited trade-off due to the small design) but most solutions are far from optimality: this suggests significant improvements possible to further reduce the number of branches explored in the enumeration DAG.

6-bit Adder	Flat	$691\mu m^2$	$0.49ns$
	CPA	$385\mu m^2$	$0.41ns$
	CLA	$415\mu m^2$	$0.36ns$
	Best Area	$366\mu m^2$	$0.38ns$
	Best Delay	$534\mu m^2$	$0.34ns$
5 bit Adder	Flat	$496\mu m^2$	$0.36ns$
	CPA	$294\mu m^2$	$0.34ns$
	CLA	$347\mu m^2$	$0.33ns$
	Best Area	$264\mu m^2$	$0.34ns$
	Best Delay	$359\mu m^2$	$0.32ns$
CSA	DesignWare	$422\mu m^2$	$0.49ns$
	Best Area	$385\mu m^2$	$0.39ns$
	Best Delay	$449\mu m^2$	$0.36ns$
4×3-bit Multiplier	Designware	$613\mu m^2$	$0.63ns$
	Best Area	$644\mu m^2$	$0.48ns$
	Best Delay	$776\mu m^2$	$0.43ns$

Table 1: Optimization results for our benchmarks.

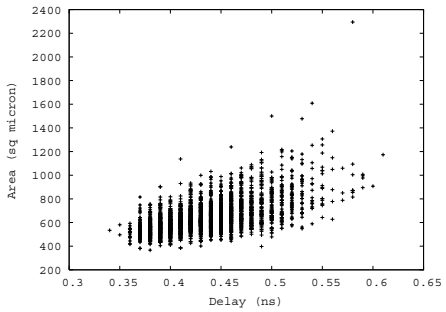

Figure 7: Area and Delay for all 6-bit adders generated by our algorithm.

Qualitatively, it is interesting to look at some of the solutions. The irregular adder of Fig. 1 (actually the fastest of the solutions generated by our algorithm), is an irregular form of carry lookahead adder *discovered* by our optimization algorithm. Similarly, in the case of the 3-input addition, our algorithm finds out automatically and without any prior knowledge the benefit of a carry-save representation. Also, by observing the circuit for the fastest multiplier output by our algorithm, we discovered peculiar dependencies among the partial product bits (as shown in Fig. 8) which can be used to improve its performance significantly. We are currently studying how to exploit such dependencies systematically, and we have designed multipliers which are almost 20% faster than those generated with the Three Greedy Approach [9].

Unsurprisingly, our main limitation is the runtime needed to achieve these results. Currently, all results except the multiplier took less than a couple of hours; larger circuits could not be optimized within a reasonable time.

8. CONCLUSIONS

We argue in this paper that research in some important cases of multiple-level logic optimization is an essential enabler to make optimization of arithmetic circuits across architectural barriers possible. Feasibility depends on our ability to enumerate efficiently all possible CSEs *of interest* across multiple logic functions. We have shown first steps in pruning the implicit enumeration DAG; by this mean, we try to obtain only useful solutions which exploit at best logic synthesizers used subsequently in the design flow. As

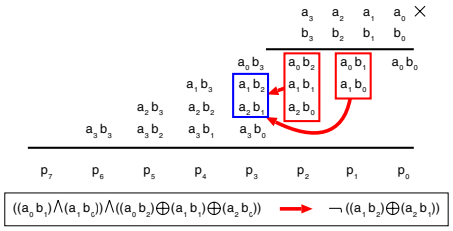

Figure 8: An illustration of complex dependencies among the partial product bits of a multiplier.

pointed out in the experimental section, this is a considerable complexity reduction in the number of solutions and in the complexity of the problem, but more work needs to be done for this type of technique to become of practical use—that is, to make it applicable to larger circuits. Our results indicate that we are still far away from producing only Pareto-optimal solutions and, among other ideas, we are considering using some branch & bound techniques to abandon early unpromising branches.

9. REFERENCES

[1] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer, New York, 1993.

[2] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[3] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, Feb. 1990.

[4] S.-C. Chang and D. I. Cheng. Efficient boolean division and substitution using redundancy addition and removing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1096–1106, Aug. 1999.

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

[6] B. D. Lee and V. G. Oklobdzija. Improved CLA scheme with optimized delay. *Journal of VLSI Signal Processing*, 3:265–74, Oct. 1991.

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

[8] A. Peymandoust and G. De Micheli. Using symbolic algebra in algorithmic level DSP synthesis. In *Proceedings of the 38th Design Automation Conference*, pages 277–82, Las Vegas, Nev., June 2001.

[9] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, C-47(3):273–85, Mar. 1998.

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

[11] A. K. Verma and P. Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 791–98, San Jose, Calif., Nov. 2004.