# Iterative Layering: Optimizing Arithmetic Circuits by Structuring the Information Flow

Ajay K. Verma [1]
ajaykumar.verma@epfl.ch

Philip Brisk [2]
philip.brisk@gmail.com

Paolo Ienne [1]
paolo.ienne@epfl.ch

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland
[2] Department of Computer Science and Engineering
University of California, Riverside, 900 University Ave. Riverside, CA 92521

## ABSTRACT

Current logic synthesis techniques are ineffective for arithmetic circuits. They perform poorly for XOR-dominated circuits, and those with a high fan-in dependency between inputs and outputs. Many optimizers, therefore employ libraries of hand-optimized arithmetic components, but cannot optimize across component boundaries. To remedy this situation, we introduce a new logic synthesis algorithm which analyzes the input circuit based on its behavior on a set of random assignments of input variables, and outputs a structural implementation of the input circuit. The method presented here is similar to the covering algorithm used in multi-level optimizations [4]; however, it is not based on *Sum-of-Product* form, or any specific input representation. Our experiments show that our approach is not only capable of automatically reproducing some known architectural implementations without any prior knowledge about the functionality of the circuit, but also, in some cases, it is able to discover completely new designs which we have not seen described in literature.

## 1. INTRODUCTION

Most arithmetic circuits have the following two properties: The first is an abundance of XOR gates, which are interspersed in the circuit. As an example consider the compressor tree, which uses full and half adders as its components. Each of two components produces two output bits, *sum* and *carry*, where the computation of sum uses solely XOR gates and the computation of carry uses AND and OR gates. Since the outputs of one component are used as an input of other component, this results in a circuit with plenty of XOR gates interspersed throughout the circuit. The second property is the high fan-in dependency between circuit inputs and outputs, e.g., in an adder the $k^{th}$ output bit depends on the $k$ least significant bits of both input operands; multipliers have similarly large support sets.

These two properties are the main reasons for the ineffectiveness of logic synthesis on arithmetic circuits. The lack of any general rule to swap the order of XOR or OR operations restricts the mobility of the two gates to a small region, barring the applicability of techniques such as height-reduction etc. The same is true for kernel and cokernel extraction algorithms used in multi-level optimization [4], which can operate on a circuit which consists of only one of the two gates, XOR and OR, along with other gates such as AND, NOT, etc.

On the other hand, to handle high fan-in dependency, *decomposition* techniques [1, 3, 2, 11, 12, 16] are used. These approaches decompose the circuit into smaller blocks. Each block usually acts as an "information compressor", meaning that it produces fewer output bits than it has input bits. After forming a block, either the complete circuit is rewritten as a function of the block output bits (in a bottom-up approach) or the inputs of the block are computed (in a top-down approach), and the process repeats. This block structure tends to reduce the number of signals at each layer of the circuit, providing it with the general structure of an inverted cone.

A decomposition is called *disjoint* if each block is decomposed into blocks with disjoint support set. Non-disjoint decompositions do not satisfy this property. Effective algorithms to compute the maximal disjoint decomposition of a circuit exist [2]. Unfortunately, many circuits exist that cannot be described effectively in terms of fully disjoint decompositions. In some cases, disjoint decompositions finds very large blocks. Finding an appropriate implementation of the large blocks is itself a major problem.

Although non-disjoint decomposition does not suffer from the above limitations, there are no known exact polynomial-time algorithms to compute optimal non-disjoint decompositions. Most non-disjoint decomposition algorithms are iterative in nature and require some form of Boolean division in each iteration to rewrite the circuit in terms of decomposed blocks. Since the result of a Boolean division is not unique, the performance of these algorithms highly depend on the chosen quotient (remainder).

In this paper, we present an algorithm which efficiently computes an appropriate architectural implementation of the input circuit. The presented algorithm, which we call *Iterative Layering*, is iterative in nature like the existing algorithms. However, it has the following characteristics which distinguish it from all the approaches we know of:

1) The presented algorithm does not rewrite the input expression after each iteration, and hence, does not use any form of division.

2) Our approach is applicable to any input representation, as long as the Boolean expression in the given representation can be efficiently converted to *Conjunctive Normal Form (CNF)*, which can be done for most representations [13].

3) Iterative Layering is based on sampling. It computes the value of the input expressions on various assignments of input variables, and analyzes it; this information is then used to find the appropriate architectural implementations of the circuit.

4) Similar to covering algorithms used in multi-level optimization [4], our algorithm uses set-cover heuristics. However, the notion of *kernels* is more general here as explained in Section 3.

Although our approach can be applied to any circuits and is not specific to arithmetic or XOR-dominated circuits, the main advantage of the approach is reflected in arithmetic circuits. This is because of the effectiveness of less complex algebraic factoring based approaches on common logic functions.

## 2. STATE OF THE ART

The performance of arithmetic circuits highly depends on their input description. For example, if a circuit of a multiplier is described in the shift-and-add fashion, synthesis tools are unable to transform it into a compressor tree based multiplier. Most of the arithmetic components are studied manually and researchers have presented manually optimized designs for specific circuits such as compressor trees [14], etc. However, the use of this approach is limited to some specific circuits, and cannot be applied to general arithmetic circuits.

Even for well studied circuits, it is not clear whether the architectural space of these circuits is completely explored. Verma *et al.* [18] have shown by using exhaustive enumeration that there exist better architectural implementations than the manually known ones, even for common circuits such as multipliers. However, due to high computational complexity of exhaustive approaches, they cannot be applied to larger circuits.

A general approach to optimize arithmetic circuits uses the high-level transformations such as those described by Verma *et al.* [17]. These transformations operate at the operation level, rather than the bit level. These transformations are used to move disparate addition and multiplication operations that occur throughout a dataflow graph to form large multi-input adders. Bit-level optimizations cannot be applied in this context, except for the specific methods used to generate the compressor trees [14].

Decomposition, introduced by Ashenhurst [1] and Curtis [3], have been a major focus in recent years. Both disjunctive [2, 11, 16] and non-disjunctive [12, 19] decomposition have been presented in the past. We have already explained that most decomposition approaches suffer from two weakness: the first is the rewriting of the original circuit after each decomposition step which requires cofactor computation; the second is the dependence of the algorithm on the input representation of the circuit. Some decomposition approaches are specific to BDDs [12, 19, 2], while some are specific to Reed-Muller form [16].

Although our approach is similar to the decomposition approaches, it does not require rewriting the circuit after each step. Also, the approach is not specific to any particular representation of input circuit. Our approach is not only able to generate known architectures for some circuits, but is also able to discover completely new architectural implementations in some cases.

## 3. MAIN IDEA

In our approach, we consider the input circuit as a unit which encodes the information about the input bits into output bits. For example, a full adder is a unit which takes three input bits and the information that it computes is the sum of the three input bits. The goal is to find an appropriate implementation of the unit, which computes the same information, but using minimum resources (i.e., time or cell area).

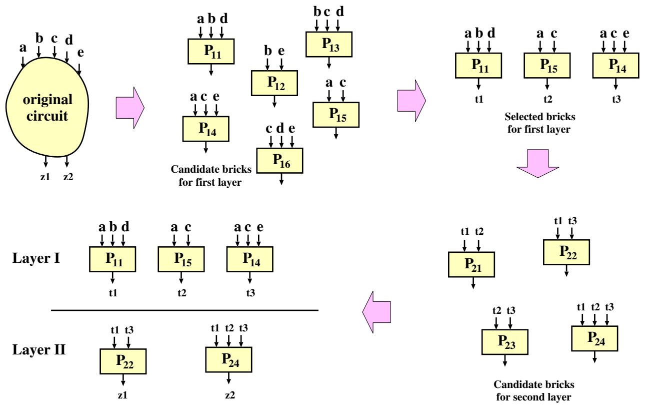


**Figure 1: An overview of the complete algorithm. At each step the algorithm computes a set of candidate bricks and then selects a subset of them based on some optimization criteria. The selected bricks form a layer and the overall structure is obtained by stacking these layers.**

Next, we define the notion of *Brick*. A brick is a small circuit, not necessarily a subcircuit of the original circuit, with $k$ inputs and a single output bit ($k$ is usually bounded by a constant). Usually the brick is used to reduce the complexity of the original circuit unit. As an example, consider the input expression $E = ac + ad + bc + bd$; using a brick $p = a + b$, the original expression can be computed like $E = pc + pd$. This reduces the number of gates from 7 to 4 (one gate to compute $p$ and 3 gates to compute $E$).

One can notice that the brick is similar to the kernel used in multi-level optimization. However, bricks are more general than the kernels. Unlike multi-level optimization, where the original expression is the OR of the product of chosen kernels and cokernels, there can be any complex functional dependency can exist between a brick and the original expression.

In our algorithm, we construct the architectural implementation of the circuit using bricks. The algorithm is iterative in nature. At each step a set of bricks is computed. Some of these are discarded immediately because they do not help in reducing the complexity of the original circuit. Among the rest, we select a subset of bricks which contains all the information about the original circuit, and is optimal with respect to some metric. The selected bricks form a layer and the layers are stacked together in order to form the overall structure of the circuit implementation. An overview of the algorithm is shown in Fig. 1. In the coming sections, we discuss the steps of the algorithm in detail. Section 4 discusses how to measure the merit of a brick. Section 5 explains the procedure for computing candidate bricks of a layer and Section 6 discusses the heuristic to select the optimal subset of bricks to form a layer.

## 4. MERIT ESTIMATION OF A BRICK

In this section we discuss how to estimate the merit of a brick. With each brick, we associate two attributes: *information fitness* and *information coverage*.

### 4.1 Information Fitness

The information fitness of a brick represents the reduction in resources to compute the original expression by using this brick. Note that, the fitness of a brick depends on the existence of other bricks. As an example consider the expression $E = ac + ad + bc + bd$, Using the brick $p_1 = a + b$ or $p_2 = c + d$ the number of gates can be reduced from 7 to 4 as discussed in the previous section. However, the introduction of brick $p_2$ in presence of $p_1$ reduces the number of gates only from 4 to 3.

Assume that we are given an input expression $E$ of $n$ input bits, a set $S$ of already chosen bricks, and a new brick $p$. The goal is to assess the reduction in the resources to compute $E$ using $p$. Unfortunately it is extremely difficult to exactly predict the usefulness of $p$ in the computation of $E$, as it will require constructing the circuit implementation of $E$ with and without using $p$. However, certain estimators can be used effectively to approximate the complexity of the circuit corresponding to a Boolean expression without actually constructing the circuit. Macii *et al.* [10] have shown that the size of the binary decision diagram corresponding to the input expression and its entropy can be used effectively to predict the complexity of the corresponding circuit. For simplicity in our approach we use the BDD size of an expression as the measure of the complexity of corresponding circuit.

Hence, we construct two BDDs, both corresponding to $E$—one which uses the $n$ input bits and the bricks in $S$ as its inputs, while the second uses the $n$ inputs and the bricks in $S \cup \{p\}$ as its inputs. Assume that the size of the two BDDs is $\alpha$ and $\beta$ respectively. We also construct the BDD corresponding to the brick $p$. If this BDD has size $\gamma$, then the use of brick $p$ reduces the BDD size from $\alpha$ to $\beta + \gamma$. We consider the ratio of two values (i.e., $\frac{\alpha}{\beta+\gamma}$) as the information fitness of the brick $p$.

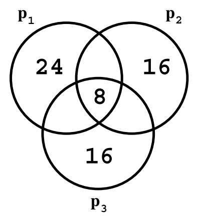
This reduces the task of computing information fitness to compute the size of the BDD of an expression whose inputs can be Boolean expressions coming from a set. Ideally, one should consider all assignments of the $n$ input variables and form a table where inputs are the original $n$-input bits along with the bricks and the target expression is $E$. The BDD can be constructed from the truth table. However, this approach is infeasible because it requires computing the complete truth table.

Instead, we consider a sample of the complete assignment space of $n$ input variables and form a partial table. At the rest of the assignments, the value of target expression is considered to be *don't care*. In order to construct the BDD from this table we use a procedure similar to the one used in the BDD package *BuDDy* [9]: We first partition the table into balanced sets according to one input, and then the two sets are partitioned recursively. This partition will result in a BDD, which can be minimized using the swapping algorithms used in BDD minimization. Bigger samples will lead to better estimations.

## 4.2 Information Coverage

In order to define the information coverage of a brick, we need to define how much information is required to define a Boolean expression. A completely specified Boolean function $f$ can be recognized by its *on-set* or *off-set*, i.e., by the partition of the Boolean space of input variables into on- and off-set. This means that, if using a set $S$ of Boolean functions we can determine whether a point lies in the on-set or off-set of $f$, then the set $S$ of Boolean functions contains at least as much information as $f$ contains. In this case, we say that there is a functional dependency [8] from the Boolean functions of $S$ to $f$.

Lee *et al.* [8] have presented a method based on SAT to decide if a functional dependency exists. It is based on the following observation: If there exist two points in the Boolean space of input variables, one in the on-set of $f$ and the other in the off-set of $f$, such that all functions in $S$ have identical values on the two points, then the value of function $f$ cannot be determined from the Boolean functions in $S$. On the other hand, if there is no such pair of points, that means corresponding to each set of values of functions in $S$ there is a unique value of $f$, i.e., the value of $f$ can be uniquely determined from the values of functions in $S$. This leads to the following necessary and sufficient condition for the existence of a functional dependency from the Boolean functions of $S$ to $f$: For each pair of points, where one point of the pair lies in the on-set of $f$, and the other point lies in the off-set of $f$, at least one function in $S$ must have non-identical values at the two points.



| $p_1$ | $p_2$ | $p_3$ | f | numPoints |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 1 | 4 |
| 0 | 1 | 1 | 0 | 2 |
| 1 | 0 | 0 | 1 | 2 |
| 1 | 0 | 1 | 0 | 4 |
| 1 | 1 | 1 | 1 | 2 |

**Figure 2: An example illustrating the relative quantities of information about a target Boolean function contained in several bricks.**

Based on the above observation we define the *information coverage* $\eta$ of a brick $p$ with respect to original circuit unit $E$ as a ratio of two integers. The denominator is the number of pairs $(x, y)$ of the Boolean space, where one point of the pair lies in the on-set of $E$ and the other point lies in the off-set of $E$. The numerator is the number of these pairs where $p$ also has different value on the two points of pair. We explain it using the following example:

$$E = a \oplus b \oplus c \oplus d,$$
$$p_1 = a \oplus b, \quad p_2 = cd, \quad \text{and} \quad p_3 = c + d.$$

Since $E = p_1 \oplus (p_3\overline{p_2})$, there is a functional dependency from $p_1, p_2, p_3$ to $E$. The function $E$ partitions the Boolean space of input variables into two sets of 8 points corresponding to its on- and off-set. This means that there are $8 \times 8 = 64$ pairs of points, each of which must be covered by one of the three bricks $p_1$, $p_2$ and $p_3$. Fig. 2 shows the exact number of pairs covered by the three bricks. Since $p_1$ covers 32 of these pairs compared to 24 pairs covered by both $p_2$ and $p_3$, $p_1$ contains more information about $E$ compared to $p_2$ and $p_3$. Although, the exact computation of the information coverage requires constructing the complete truth table of the original function $E$, the following theorem, whose proof we omit due to lack of space, says that the information coverage of a brick can be estimated very accurately by choosing a moderate size sample of the complete truth table.

THEOREM 1. *Let the on- and off-set of an expression $E$ be $(A \cup C, B \cup D)$, and the on- and off-set of a brick $p$ be $(A \cup B, C \cup D)$. The size of sets $A, B, C, D$ are $a, b, c, d$ respectively. If a sample of size $N$ is taken from the complete truth table, which contains at least one point of each set $A, B, C, D$, and $\eta$ is the information coverage of $p$ with respect to $E$ on this sample, then*

$$\mathbf{E}(\eta) = \frac{ad + bc}{(a + c)(b + d)},$$
$$\mathsf{Var}(\eta) \approx \frac{a + b + c + d}{(a + c)^2(b + d)^2} \cdot \left( \frac{ad(a + d) + bc(b + c)}{N} + \frac{(a + b + c + d)(ad + bc)}{N^2} \right).$$

Note that the actual information coverage of $p$ is $\alpha = \frac{ad+bc}{(a+c)(b+d)}$, hence, the theorem says that using a sample of size $N$, the computed information coverage will lie in the region $\left(\alpha \pm \theta(\frac{1}{N^{0.5}})\right)$ with very high probability.

## 5. BRICKS AND THEIR COMPUTATION

In this section we describe how to compute the bricks introduced in the previous section. An ideal brick must not have a large number of inputs for two reasons: First, finding an appropriate implementation of the circuit corresponding to a large expression would not be easy. Second, there are more candidate bricks with higher number of inputs; in fact, enumeration based approaches will not work for bricks with more than 6 inputs. On the other hand, the information fitness of a small brick is usually small, meaning it would be difficult to judge the superiority of one brick over another. This forces the size of bricks to be as large as possible. In our approach, we consider only bricks with $k$ or fewer inputs, where $k$ is usually smaller than 6.

In order to compute the set of candidate bricks for an expression $E$ with $n$ inputs, we consider all combination of $k$ input bits, and find the bricks with the chosen set of input bits. For a particular combination of input bits, the bricks are computed by sampling the assignment space of the remaining $(n-k)$ inputs. The main idea behind the sampling is explained below.

Let $B$ be the set of bits for which we wish to compute bricks and $R$ be the remaining bits. One possibility, which runs in exponential time, is to enumerate all cofactors of $E$ with respect to variables in $R$ and store them in a set $S$. $S$, effectively, contains the bricks, because all information regarding the bits in $B$ is contained in $S$. We explain this with the use of an example:

$$\begin{aligned} &E = ab \oplus cd \oplus (a \oplus b)(c \oplus d),\\ &B = \{a, b, c\}, \quad R = \{d\}. \end{aligned}$$

In this case, $R$ contains only a single variable, hence corresponding to its two assignments, we can get two elements of $S$, which are the cofactors of $E$ with respect to $d$.

$$S = \{E_d, E_{\overline{d}}\} = \{ab \oplus bc \oplus ac, ab \oplus bc \oplus ac \oplus a \oplus b \oplus c\}.$$

Hence, there will be two bricks corresponding to the variables $a, b, c$. A naive implementation of the above approach does not work because it suffers from the three main problems. Next, we discuss these problems one by one and our approach to handle them.

### 5.1 High Computational Complexity

The first problem is the exponential complexity of the approach: if there are $r$ elements in $R$, then we need to consider all $2^r$ assignments of these variables. This could be computationally infeasible for large $r$. Although there are $2^r$ cofactors, all of them are Boolean expressions of $k$ variables, where $k$ is usually small (never exceeding 6). This means that most of these cofactors can be expressed as a function of other cofactors, i.e., the number of irredundant cofactors is usually very small. Hence, in order to find the irredundant cofactors we use sampling rather than computing all $2^r$ cofactors. Consider the following example:

$$\begin{aligned} &E = (ab + bc + ac) \oplus de \oplus (a \oplus b \oplus c)(d \oplus e),\\ &B = \{a, d\}, \quad R = \{b, c, e\}. \end{aligned}$$

In this case, there are only two irredundant cofactors, i.e., $S = \{ad, a + d\}$. The first cofactor can be obtained by considering any of the four assignments $(000, 011, 101, 110)$ of variables in $R$, while the second cofactor can be obtained by considering any of the four assignments $(001, 010, 100, 111)$. This means a random sampling of the complete assignment space of $R$ is highly likely to find the irredundant cofactors without considering all assignments of variables in $R$.

However, in some cases the random sampling may miss some irredundant cofactors. For example, consider the following expression which computes the carry output of a four-bit adder:

$$\begin{aligned} &E = a_3b_3 + (a_3 + b_3)a_2b_2 + (a_3 + b_3)(a_2 + b_2)a_1b_1 +\\ &\quad (a_3 + b_3)(a_2 + b_2)(a_1 + b_1)a_0b_0, where\\ &B = \{a_0, b_0\}. \end{aligned}$$

There is only one assignment of variables $a_3, b_3, a_2, b_2, a_1, b_1$ for which $E$ actually depends on bits $a_0$ and $b_0$; random sampling is quite likely to miss this cofactor.

In our approach, we perform random sampling but also include an analytical model that helps us to find missing irredundant cofactors. Let $S$ be the set of cofactors found using random sampling; we want to determine if all the required information about the input bits $B$ is contained in $S$. This is effectively the same as deciding if $E$ is functionally dependent on the expressions in $S \cup R$.

First, we add the input bits in $R$ to $S$ and determine if there is a functional dependency from the functions of $S$ to $E$ using Lee *et al.*'s approach [8]. Suppose that the variables in $B$ are $b_1, b_2, \ldots, b_m$, and the variables in $R$ are $r_1, r_2, \ldots, r_n$. We introduce two sets of *dummy variables* $c_1, c_2, \ldots, c_m$ and $s_1, s_2, \ldots, s_n$. The original and dummy variables correspond to two assignments of the original variables. Since all expressions in $S$ are identical in both cases, we introduce the following constraints:.

$$\begin{aligned} r_i = s_i, &\quad 1 \le i \le n,\\ e_j(b_1, b_2, \ldots, b_m) = e_j(c_1, c_2, \ldots, c_m), &\quad 1 \le j \le t\\ E(b_1, \ldots, b_m, r_1, \ldots, r_n) \neq E(c_1, \ldots, c_m, s_1, \ldots, s_n). \end{aligned}$$

Here, $t$ is the number of irredundant cofactors $e_j$'s in $S$. These constraints will be satisfied if and only if there exist two assignments of variables on which all expressions in $S$ have identical values, but $E$ has different values, i.e., $E$ is not functionally dependent on the Boolean expressions in $S$. Satisfying assignments also provide the missing irredundant cofactors.

This application of SAT does not require sampling; however, omitting sampling will require finding all satisfying assignments of SAT, which can be exponentially many. We use random sampling to reduce the number of satisfying assignments in the SAT instance to a small number.

### 5.2 Pruning the Set of Cofactors

In some cases, enumerating the assignments of bits in $R$ produces some cofactors which contain no meaningful information that helps to compute $E$. For example, consider the expression $E = (a_0b_1 + a_1b_0)$ and $B = \{a_0, b_0\}$. One of the cofactors of $E$, which is obtained by the assignment $a_1 = b_1 = 1$, is $(a_0 + b_0)$. Since knowing the value of this cofactor does not help in computing the value of the original expression, it should be discarded. To check for these cases, we consider each cofactor as a brick and estimate its information fitness using the estimator discussed in Section 4.1. The cofactors whose corresponding bricks have information fitness below a threshold are discarded.

### 5.3 Brick Computation from Cofactors

Since the irredundant cofactors computed in previous section represent the information about the chosen set of input bits, the bricks must be chosen in such a way that they should be able to generate all of the cofactors. In order to find the best set of bricks corresponding to the set of cofactors, we use our algorithm Iterative Layering on the expressions corresponding to cofactors with even smaller bricks. This results in an implementation of the cofactors using input bits. The signals crossing any cut of the resulting circuit correspond to a potential set of bricks, as they contain all the information to compute the cofactors. We consider each of the cuts

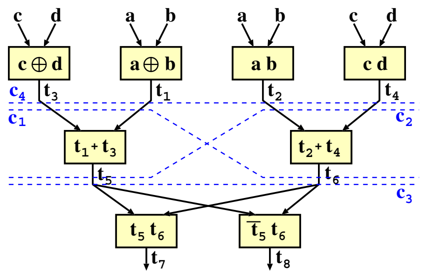


**Figure 3: An example showing how the bricks are generated from the generated set of irredundant cofactors.**

```
// The function takes a set $\mathcal{P}$ of pairs of
// points, a set of bricks, and returns an optimal set
// of bricks covering all pairs of $\mathcal{P}$.
brickSelection (Set $\mathcal{P}$, Set I) {
Set C = $\phi$
  do {
    // At each step a brick is chosen for which the product
    // of information fitness and coverage is maximum.
    p = chooseBestBrick (I);
    C = C $\cup$ {p};
    I = I - {p};

    // After the addition of each new element in C, the
    //attributes, of the bricks in I are modified.
    modifyAttributes (I, C);
  while(all elements of $\mathcal{P}$ are not covered);
  return C;}
```

**Figure 4: A heuristic to solve the brick selection algorithm based on Johnson's greedy heuristic to solve the set cover.**

and the one whose signals correspond to the best set of bricks in terms of cumulative information fitness is chosen. Since all cofactors are made of $k$ variables ($k \leq 6$), the DAG corresponding to the implementation of cofactors is small, resulting only in a few cuts, never exceeding 100 in our experiments. For example, consider the following expression:

$$E = ab(c \oplus d \oplus e) + cd(a \oplus b \oplus e),$$
$$B = \{a, b, c, d\}, \quad R = \{e\},$$
$$S = \{ab(c \oplus d) + cd(a \oplus b), ab(c\overline{\oplus}d) + cd(a\overline{\oplus}b)\}.$$

When we apply our algorithm Iterative Layering to find an implementation of the Boolean expressions in $S$ with smaller bricks (with 2 inputs), it finds the following implementation.

$$t_1 = a \oplus b, \quad t_2 = ab, \quad t_3 = c \oplus d, \quad t_4 = cd,$$
$$t_5 = t_1 + t_3, \quad t_6 = t_2 + t_4, \quad t_7 = t_5 t_6, \quad t_8 = \overline{t_5} t_6,$$
$$S = \{t_7, t_8\}.$$

Fig. 3 shows various cuts of the implementation DAG. The best cut is $C_3$, which is crossed by signals $t_5$ and $t_6$. Hence, the chosen bricks will be $t_5$ and $t_6$. Using these bricks the original expression $E$ can be computed like $E = t_6(t_5 \oplus e)$.

## 6. BRICK SELECTION HEURISTIC

In this section, we discuss how to select bricks for a layer. The input is a Boolean expression $E$ and a set of bricks satisfying the property that $E$ can be written as a function of the bricks. The goal is to find a subset of bricks such that there is functional dependency from the chosen set of bricks to $E$, and the cumulative information fitness of the chosen bricks is maximum.

We consider all assignments of the input variables, and form a table $T$ which contains the values of bricks and the input expression on the assignments of input variables. In this case, the functional dependency of $E$ on a set of bricks can be verified using the criteria explained in Section 3. This means that the problem can be reduced to a problem similar to *Set Cover* [5], The universe correspond to the set $\mathcal{P}$ of all pair of points, where one point of each pair lies in the on-set of $E$, and the other point lies in the off-set of $E$. The individual sets of each brick $p$ correspond to the set of these pairs which are covered by $p$. The goal is to cover the universe using a subset of bricks whose cumulative fitness is maximum.

Set cover is NP-hard [5] and is solved using polynomial time heuristics. We use a slight variation of the greedy heuristic proposed by Johnson [6] which is shown in Fig. 4. The heuristic is iterative in nature, it starts with an empty collection of bricks, and in each iteration it includes a brick for which the product of information fitness and information coverage is maximum. The information fitness and information coverage of remaining bricks are modified, and the algorithm continues.

The problem with this approach is the exponentially large size of table $T$. Hence, instead of considering all assignments of input variables, we choose a sample of the complete assignment space of the input variables. This results in fewer number of pairs in $\mathcal{P}$, which needs to be covered by the bricks. The algorithm of Fig. 4 is applied on this smaller set $\mathcal{P}$. After finding a set of bricks we check if there exist a functional dependency from the chosen set of bricks to the input expression $E$. This is done by solving an instance of SAT as explained in the earlier sections. If there does not exist such functional dependency, then the SAT solver returns a pair of points which is not covered by any of the chosen bricks. The two points of the pair are added in the table and the whole algorithm is repeated. The same sample is used to compute the bricks of next layer.

Note that information fitness of bricks is independent of the chosen sample in $T$, while the preciseness of information coverage of a brick depends on the size of chosen sample as discussed in Theorem 1. A large sample is preferred. In order to be able to use larger sample we must reduce the complexity of the algorithm presented in Fig. 4. Next, we will discuss how to improve the complexity of algorithm presented in Fig. 4.

From now on, by the on- and off-set of a function we mean the on- and off-set of the function restricted to the chosen sample. Let us assume that $N$ is the size of sample (i.e., number of rows in the table $T$), $m$ is the total number of bricks, and $r$ is the number of information bricks in the set output by the brick selection algorithm. In the algorithm of Fig. 4 after each iteration the fitness of each remaining brick is estimated, this will result in $O(mr)$ number of calls to the estimator function, and cannot be avoided. However, the computation of information coverage can be improved.

The number of pairs in $\mathcal{P}$ can be $O(N^2)$, also each pair can be covered by any number of bricks, hence, the cumulative size of all sets corresponding to bricks will be $O(mN^2)$. In each iteration a brick is chosen, and sets corresponding to remaining bricks should be modified by removing the already covered pairs from them. This can be done in $O(mN^2)$, hence, the overall complexity of modifying the sets will be $O(mrN^2)$ resulting in the overall complexity $O(mr(N^2 + T_{estimator}))$ of the brick selection algorithm.

### 6.1 Complexity Reduction via Implicit Representation

We use the specific structure of the sets corresponding to a brick in order to reduce the complexity of brick selection algorithm. For a brick $p$ consider the partition of the rows of table into four sets: $A$, $B$, $C$ and $D$ such that $(A \cup C, B \cup D)$ is the partition of the rows

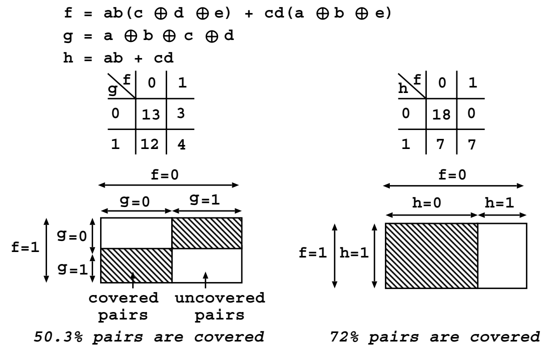
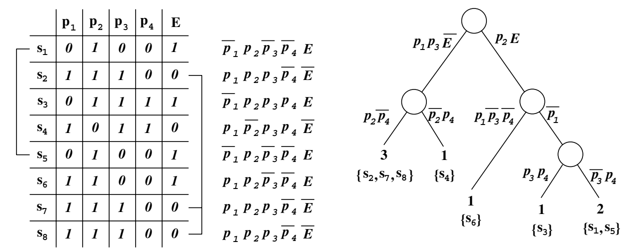


**Figure 5: The pairs covered by bricks can be described using a compact implicit representation.**



| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $E$ | |
|---|---|---|---|---|---|---|
| $s_1$ | 0 | 1 | 0 | 0 | 1 | $\overline{p}_1 p_2 \overline{p}_3 \overline{p}_4 E$ |
| $s_2$ | 1 | 1 | 1 | 0 | 0 | $p_1 p_2 p_3 \overline{p}_4 \overline{E}$ |
| $s_3$ | 0 | 1 | 1 | 1 | 1 | $\overline{p}_1 p_2 p_3 p_4 E$ |
| $s_4$ | 1 | 0 | 1 | 1 | 0 | $p_1 \overline{p}_2 p_3 p_4 \overline{E}$ |
| $s_5$ | 0 | 1 | 0 | 0 | 1 | $\overline{p}_1 p_2 \overline{p}_3 \overline{p}_4 E$ |
| $s_6$ | 1 | 1 | 0 | 0 | 1 | $p_1 p_2 \overline{p}_3 \overline{p}_4 E$ |
| $s_7$ | 1 | 1 | 1 | 0 | 0 | $p_1 p_2 p_3 \overline{p}_4 \overline{E}$ |
| $s_8$ | 1 | 1 | 1 | 0 | 0 | $p_1 p_2 p_3 \overline{p}_4 \overline{E}$ |

**Figure 6: A tree based storage of sample can be used to avoid the storage of pairs covered by each brick.**

into on- and off-set of $E$, while $(A \cup B, C \cup D)$ is the partition of the rows into on- and off-set of $p$. The set $\mathcal{P}$ corresponds the Cartesian product of $(A \cup C)$ and $(B \cup D)$, i.e., $(A \cup C) \times (B \cup D)$. This can be expanded like this:

$$\mathcal{P} = (A \cup C) \times (B \times D),$$
$$\mathcal{P} = (A \times D) \cup (C \times B) \cup (A \times B) \cup (C \times D).$$

Note that, each pair in $(A \times D)$ as well as in $(C \times B)$ is covered by the brick $p$, while none of the pairs in $(A \times B)$ and $(C \times D)$ is covered by $p$. Hence, the set of pairs corresponding to brick $p$ can be represented like $(A \times D) \cup (C \times B)$. An example of this representation is shown in Fig. 5. Using this representation, the universal set $\mathcal{P}$ of pairs, as well as the sets corresponding to each brick can be generated in $O(mN)$ time by considering all the rows of table one by one, and put them in the on- or off-set of each brick. The computation of pairs covered by a particular brick requires computing the intersection of the on- and off-set of this brick with the on- and off-set of $E$ (in order to extract $A, B, C, D$ from $A \cup C, B \cup D, A \cup B, C \cup D$).

At the beginning of the algorithm shown in Fig. 4, the pairs corresponding to one Cartesian product $((A \cup C) \times (B \cup D))$ are to be covered. After choosing a brick $p$ the pairs in $(A \times D) \cup (C \times B)$ are already covered, and the pairs corresponding to two Cartesian products $(A \times B)$ and $(C \times D)$ needed to be covered. The number of Cartesian products to be covered may increase after each iteration. In general, there can be $\kappa$ Cartesian products to be covered which are $A_1 \times B_1, A_2 \times B_2, \ldots, A_\kappa \times B_\kappa$ for pairwise disjoint $A_i$'s and $B_j$'s. Since the union of $A_i$'s must be a subset of the on-set of $E$, it means $\kappa$ will never exceed $O(N)$.

If the on- and off-set of a new brick are $X$ and $Y$, then the elements of a particular Cartesian product $(A_i \times B_i)$ covered by this brick will be $((A_i \cap X) \times (B_i \cap Y)) \cup ((A_i \cap Y) \times (B_i \cap X))$. Hence, all the pairs covered by this brick can be represented as the union of $((A_i \cap X) \times (B_i \cap Y)) \cup ((A_i \cap Y) \times (B_i \cap X))$ for various $i$'s. If this brick is chosen, then the remaining pairs to be covered can be represented as the union of $((A_i \cap X) \times (B_i \cap X)) \cup ((A_i \cap Y) \times (B_i \cap Y))$ for various $i$. Since, all sets are sorted, this can be achieved in linear time, i.e., in $O(N + \kappa) = O(N)$ time. This will result in the overall complexity $O(mr(N + T_{estimator}))$ of brick selection algorithm, which is better by a factor of $N$ compared to the naive implementation.

## 6.2 Space Complexity Reduction based on overlapped computation

In the above approach, we need to store the set of covered pairs corresponding to each brick. Here, we discuss a way to avoid this extra storage as well as to reduce the average case time complexity. Assume that during the execution of the brick selection algorithm $l$ bricks have already been selected, whose on- and off-sets are $(A_1, B_1) \ldots (A_l, B_l)$. Also assume that the on- and off-set of $E$ are $A_0$ and $B_0$. In this case the number of new pairs covered by a new brick $p$ with on-set $X$ and off-set $Y$ will be the ones which are not covered by any of the chosen bricks. The number of such pairs can be given by the sum of $\mid A_0 \cap R_1 \cap \ldots \cap R_l \cap X \mid \times \mid B_0 \cap R_1 \cap \ldots \cap R_l \cap Y \mid$ and $\mid A_0 \cap R_1 \cap \ldots \cap R_l \cap Y \mid \times \mid B_0 \cap R_1 \cap \ldots \cap R_l \cap X \mid$, where each of the $R_i$ is either $A_i$ or $B_i$. Note that, although there are $O(2^l)$ terms in the summation, many of them are zero. In fact, at most $O(N)$ of these terms can be non-zero. Hence, it is important to consider only the non-zero term. These sums are computed for each remaining brick.

Next, we describe how to do this computation without storing $A_i$ and $B_i$'s. First we partition the chosen sample into sets, such that for all points in a set the values of all bricks and of the original expression $E$ are unique. Since, for each set the values of $E$ along with all bricks are fixed, the only thing important about a set is its size and the value of $E$ and bricks on the points of this set, which can be called a *signature* of the set. This means, we have a set of $(m+1)$-bit signatures each having a number, which is the size of the corresponding partitioned set. We store these signatures as the leaves of a balanced binary tree as shown in Fig. 6. Since, all the signatures are $(m+1)$-bit wide, the maximum height of the resulting tree can be $(m+1)$. On the other hand, since there at at most $N$ signatures, a balanced partitioning may result in a tree with height $O(lgN)$.

During the execution of the brick selection algorithm, we needed to compute the aforementioned sums for all remaining bricks. We compute these sums by traversing the tree. In the beginning all sums are assumed to be zero, during the traversal these sums are incremented one by one. First we partition the leaves of the above constructed tree into sets corresponding to $R_0 \cap R_1 \cap \ldots \cap R_l$, where $R_0$ is the either the on- or off-set of $E$, and each of the remaining $R_i$ is either the on- or off-set of $i^{th}$ chosen brick. This can be done by considering all leaves individually.

Next, we traverse all paths from leaves to root one by one. Assume that a path starts from a leaf which belongs to the set $R_0 \cap R_1 \cap \ldots \cap R_l$. In the path if we come across an edge which is annotated by $p$ for one of the remaining bricks, that means this leaf belongs to the $R_0 \cap R_1 \cap \ldots \cap R_l \cap X$, where $X$ is the on-set of $p$, while if the edge is annotated with $\overline{p}$, then this leaf belongs to the $R_0 \cap R_1 \cap \ldots \cap R_l \cap Y$, where $Y$ is the off-set of $p$, and must be added to the corresponding set. The overall worst case time complexity still remains the same $O(mr(N + T_{estimator}))$, but in practice the algorithm works significantly faster than the previous implementation.

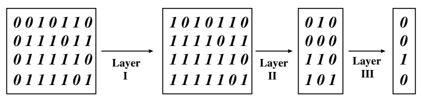


**Figure 7: Example execution of the max84 circuit, generated by Iterative Layering.**

## 7. EXPERIMENTAL RESULTS

We have implemented the *Iterative Layering* algorithm in C++. SAT instances with fewer than 20 variables are solved by exhaustive enumeration of all possible assignments of input variables; SAT instance with 20 or more variables are solved using the BDD package *BuDDy* [9].

The input expressions are represented in the *blif* format, and the output is a VHDL description of the optimized circuit. We compare the circuits produced by Iterative Layering to circuits produced using Progressive Decomposition [16], and library or manually optimized implementations of each circuit using known good designs from prior literature. All of the circuits are synthesized using a common standard cell library for UMC $90nm$ CMOS technology, with the optimization strategy to minimize delay.

The benchmarks are divided into three categories: basic arithmetic components, MCNC benchmarks, and composite arithmetic circuits. Table 1 shows the results of our experiments.

### 7.1 Arithmetic Components

The first arithmetic component is a 16-bit *Ripple Carry Adder*. Both our algorithm and Progressive Decomposition transform it into an implementation similar to that of *Carry Lookahead Adder* which has almost the same performance as the library implementation. The direct synthesis of the manually optimized circuit yields similar results.

The second component is a 12-bit 3-input adder built from two *Ripple Carry Adders* in sequence. Due to the mixture of XOR and OR gates, techniques based on algebraic factorization fail to optimize this circuit, which is reflected by the poor performance of the synthesis of original circuit; however, both Iterative Layering and Progressive Decomposition transform the first adder into a carry-save adder to compute the sum more efficiently.

The next three circuits are $8 \times 8$, $9 \times 9$, and $10 \times 10$-bit shift-and add multipliers. Direct implementation of these circuits results in a higher delay and area compared to a library implementation, which uses a standard partial product generator followed by a Wallace-style compressor tree for partial product reduction. Progressive Decomposition cannot proces these circuits because of exorbitantly large input expressions in *Reed-Muller* Form. Iterative Layering, in contrast, automatically converts the shift-and-add multiplier into a Wallace-style design that uses a compressor tree, achieving comparable delay and area to the optimized library implementation.

Max84 computes the maximum of four 8-bit integers. The naive implementation uses a binary tree of comparators. Progressive decomposition was unable to optimize this circuit. Iterative Layering, however, was able to automatically infer a better circuit structure which is $40\%$ faster than the original. Fig. 7 illustrates the execution of the new circuit. To the best of our knowledge, no manual designer has implemented this particular circuit architecture before. We have verified its correctness by considering every possible combination of input bits. We are currently analyzing the resulting architecture and its extension for larger bitwidth integers.

### 7.2 MCNC Circuits

The first MCNC benchmark was 9symml. Compared to the naive implementation, Progressive Decomposition improved the critical path delay by $7\%$, but increased the area; Iterative Layering, in contrast, achieved a reduction in critical path delay of $11\%$, coupled with a reduction in area of $26\%$. For the next three MCNC benchmarks, sqrt8, rd64 and rd73, the commercial synthesis tool's optimization strategy was fairly effective and Iterative Layering was unable to improve them.

For the last MCNC benchmark, t481, Progressive Decomposition and Iterative Layering both improved the critical path by $50\%$ and the area by $60\%$ compared to the commercial synthesis tool; this result indiciates that these decomposition methods have the potnetial to significantly improve upon the state-of-the-art.

### 7.3 Compound Circuits

Next, we turn our attention to three compound arithmetically-intensive circuits. Progressive Decomposition was unable to optimize any of them because they yielded very large representations when converted to Reed-Muller Form. The first compound circuit is taken from the adpcmcoder benchmark [7] benchmark and computes $\frac{(\mathsf{delta}+0.5)\times\mathsf{step}}{4}$, where delta is a 3-bit integer and step is a 16-bit integer. The manual implementation was produced by clustering the four additions nodes together according to the transformations described by Verma *et al.* [17] to form a compressor tree. Our approach, in contrast, was able to effectively infer these transformations, but at the bit-level, rather than the operation level.

The second composite circuit is taken from the G721 benchmark [7] and corresponds to a customized 11-bit floating-point multiplier. Iterative Layering reduced the critical path delay by $12\%$, while increasing the area by $55\%$. The third composite circuit is a *Sum-of-Absolute Differences (SAD)* unit used in a variety of video coding algorithms such as H.264. The manually optimized implementation was suggested by Vassiliadis *et al.* [15], and is optimized for area, rather than delay. Compared to the commercial synthesis tool, Iterative Layering improves the critical path delay by $18\%$.

### 7.4 Discussion

The primary goal of Iterative Layering is to effectively improve naive implementations of known circuits into known good implementations. Iterative Layering was particularly successful in this respect with the addition and multiplication components. Given its ability to find good implementations of well-understood circuits, we hoped to be able to use it for circuits that are less well-understood. In this case, Iterative Layering was also successful, as it automatically found an architecture for the Max84 benchmark which, to the best of the authors' knowledge, has not been discovered before. In the future, we hope to use Iterative Layering to help us design new arithmetic components for use in larger applications.

The runtime of Iterative Layering depends on the size of SAT instances which are generated during its execution. For the circuits listed in Table 1, the longest running benchmark was adpcmcoder, which required approximately 20 minutes.

## 8. CONCLUSIONS AND FUTURE WORK

Iterative Layering has been presented as a pre-synthesis optimization strategy for a wide variety of circuits. Unlike prior decomposition algorithms, Iterative Layering does not rewrite the circuit after computing a layer and is not tied to any specific circuit representation. We have emphasized arithmetic circuits in our experiments because there is strong evidence that they present a formidable challenge for commercial synthesis tools. Our experiments show that Iterative Layering has the ability to automatically infer known circuit implementations from prior literature on hand-optimized arithmetics.

| Benchmark | Original | | Progressive Decomposition | | **Our Algorithm** | | Library / Manual Optimization | |
|---|---|---|---|---|---|---|---|---|
| | Delay (ns) | Area ($\mu m^2$) | Delay (ns) | Area ($\mu m^2$) | **Delay (ns)** | **Area** ($\mu m^2$) | Delay (ns) | Area ($\mu m^2$) |
| 16-bit Adder | 0.24 | 3152 | 0.26 (+ 8%) | 2966 (– 6%) | 0.24 ( **0%**) | 3960 (**+ 25%**) | 0.23 (– 4%) | 2030 (– 35%) |
| 12-bit Ternary Adder | 0.64 | 2776 | 0.33 (– 48%) | 3826 (+ 38%) | 0.33 (**– 48%**) | 3930 (**+ 41%**) | 0.31 (– 51%) | 3676 (+ 32%) |
| 8 × 8-bit Multiplier | 0.93 | 7610 | – | – | 0.55 (**– 40%**) | 6047 (**– 21%**) | 0.57 (– 38%) | 6200 (– 18%) |
| 9 × 9-bit Multiplier | 1.09 | 8986 | – | – | 0.57 (**– 48%**) | 8198 (**– 9%**) | 0.60 (– 45%) | 7589 (– 16%) |
| 10 × 10-bit Multiplier | 1.27 | 11637 | – | – | 0.62 (**– 51%**) | 8570 (**– 26%**) | 0.67 (– 47%) | 8087 (– 31%) |
| Max84 | 0.46 | 1755 | – | – | 0.28 (**– 40%**) | 1331 (**– 24%**) | – | – |
| 16-bit, 4-stage Barrel Shifter | 0.14 | 5408 | 0.16 (+ 14%) | 5504 (+ 2%) | 0.14 ( **0%**) | 5395 ( **0%**) | 0.13 (– 7%) | 5900 (+ 9%) |
| 9symml | 0.28 | 1608 | 0.26 (– 7%) | 2312 (+ 44%) | 0.25 ( **– 11%**) | 1195 ( **– 26%**) | – | – |
| sqrt8 | 0.14 | 614 | 0.18 (+ 29%) | 722 (+ 18%) | 0.14 ( **0%**) | 614 ( **0%**) | – | – |
| rd84 | 0.22 | 1035 | 0.24 (+ 9%) | 883 (– 15%) | 0.22 ( **0%**) | 921 ( **– 11%**) | – | – |
| rd73 | 0.17 | 674 | 0.16 (– 6%) | 917 (+ 36%) | 0.16 ( **– 6%**) | 1061 ( **57%**) | – | – |
| t481 | 0.36 | 997 | 0.18 (– 50%) | 402 (– 60 %) | 0.18 ( **–50 %**) | 402 ( **– 60%**) | – | – |
| adpcmcoder [7] | 0.85 | 5678 | – | – | 0.46 (**– 46%**) | 5121 (**– 10%**) | 0.49 (– 42%) | 4901 (– 14%) |
| G721 [7] (fmult) | 0.94 | 5142 | – | – | 0.82 (**– 12%**) | 7998 (**+ 55%**) | – | – |
| Sum-of-Absolute Differences (SAD) [15] | 0.61 | 5377 | – | – | 0.50 (**– 18%**) | 11503 (**+ 113%**) | 0.78 (+ 28%) | 3913 (– 27%) |

**Table 1: Results of our arithmetic optimizations on all benchmarks. The – on several entries mean that the corresponding algorithm ran out of time, or was unable to find an implementation of the circuit.**

The runtime of Iterative Layering, at present, is still a concern, primarily due to the limitations of the used SAT solver. In the future, we intend to switch to more advanced SAT solvers and to restructure the algorithm to reduce the number of calls to SAT.

## 9. REFERENCES

[1] R. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, pages 74–116, Cambridge, Mass., Apr. 1957.

[2] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *Proceedings of the International Conference on Computer Aided Design*, pages 78–82, San Jose, Calif., Nov. 1997.

[3] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Von Nostrand, Princeton, N.J., 1962.

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

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[6] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, Dec. 1974.

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

[8] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental sat solving. In *Proceedings of the International Conference on Computer Aided Design*, pages 227–33, San Jose, Calif., Nov. 2007.

[9] J. Lind-Nielsen. *BuDDy: A Binary Decision Diagram Package*. Version 2.4, `http://buddy.sourceforge.net/`.

[10] A. Macii, E. Macii, G. Odasso, M. Poncino, and R. Scarsi. Regression-based macromodelling for delay estimation of behavioral components. In *Proceedings of the 9th Great Lakes Symposium on VLSI*, pages 188–91, Ann Arbor, Mich., Mar. 1999.

[11] S.-i. Minato and G. De Micheli. Finding all simple disjunctive decompositions using irredundant sum-of-products forms. In *Proceedings of the International Conference on Computer Aided Design*, pages 111–17, San Jose, Calif., Nov. 1998.

[12] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proceedings of the 38th Design Automation Conference*, pages 103–8, Las Vegas, Nev., June 2001.

[13] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, Sept. 1986.

[14] 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.

[15] S. Vassiliadis, E. A. Hakkennes, J. S. S. M. Wong, and G. G. Pechanek. Sum-absolute-difference motion estimation accelerator. In *Proceedings of the 24th EUROMICRO Conference*, pages 559–66, Vesteras, Sweden, Aug. 1998.

[16] A. K. Verma, P. Brisk, and P. Ienne. Progressive decomposition: A heuristic to structure arithmetic circuits. In *Proceedings of the 44th Design Automation Conference*, pages 404–9, San Diego, Calif., June 2007.

[17] A. K. Verma, P. Brisk, and P. Ienne. Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-27(10):1761–74, Oct. 2008.

[18] A. K. Verma and P. Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 445–50, San Francisco, Calif., July 2006.

[19] C. Yang and M. Ciesielski. BDS: A BDD-based logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(7):866–876, July 2002.