A Decomposition Algorithm to Structure Arithmetic Circuits

Ajay K. Verma
AjayKumar.Verma@epfl.ch

Philip Brisk
Philip.Brisk@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

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

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. Multipliers are particularly challenging because of the high fanout that results from the partial product generator. 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 that analyzes the cofactors of the Boolean input expressions. To the best of our knowledge, this approach is the first general logic synthesis algorithm that can handle complex arithmetic circuits such as multipliers of non-trivial size without having any prior knowledge about the functionality of the circuit; it can also optimize larger composite arithmetic circuits that contain a variety of components. The approach reduces the delay of the composite circuits by 15–40% compared to the use of locally optimized library components without cross-component optimization.

1. INTRODUCTION AND MOTIVATION

Logic synthesis for arithmetic circuits remains an unsolved problem. Although specific algorithms exist for specific circuits such as multipliers etc., logic synthesis methods based on algebraic factoring are unable to convert a naive description of these circuits into an implementation with performance similar to the manually designed circuits. Most optimizers employ libraries of manually designed arithmetic components. Arithmetic operations in a large circuit are replaced by components from the library, reducing the complexity of the synthesis process; however, this approach has three significant drawbacks. Firstly, optimization across component boundaries is limited, and affects the architectural implemenetation of the component minimally; secondly, the architectural choices are limited to the components in the library; and thirdly, the designer's choice of operations dictates the point in the circuit where each library component is inserted.

Two factors have historically limited the effectiveness of logic synthesis tools when applied to arithmetic circuits. Firstly, most arithmetic circuits are dominated by XOR operations, which are non-monotonic; this makes covering-based techniques inapplicable for these types of circuits. Secondly, high fan-in dependencies between inputs and outputs, i.e., most of the output bits depend on a large fraction of the input bits, inhibits decomposition. For example, in an adder, the k^{th} least significant output bit depends on the k least significant bits of both input operands; multipliers have similar issues.

Historically, decomposition [2, 8, 4, 14, 15]. has been used to break high fan-in dependencies. This approach decomposes each circuit into smaller blocks. Each block usually acts as a compressor, meaning that it produces fewer output bits than 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 blocks 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. Fig. 1 shows how this decomposition can impose this type of structure on a circuit.

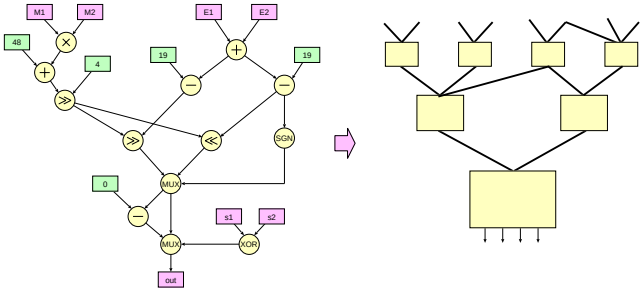

Figure 1: An illustration of circuit decomposition, which form the structure of the circuit. Small blocks can be optimized using exhaustive method.

The various blocks in a decomposition correspond to common subexpressions of a circuit. This is the main reason why circuit decomposition effectively reduces high fan-in dependencies. In fact, the decomposition effectively converts a disjoint collection of Boolean expressions into a directed acyclic graph (DAG) that implements the circuit. Unlike the DAGs corresponding to original expressions, where only primary input nodes have multiple fanout, in the new DAG the intermediate nodes which are blocks may also have multiple fanout. Furthermore, if the blocks are small in terms of the numbers of inputs and outputs, then exhaustive architectural exploration algorithms [22] can enumerate the Pareto-optimal implementations of each block in terms of delay and area.

Mishchenko *et al.* [15] presented a classification scheme to provide a better understanding of different approaches to circuit decomposition. The *support* of a block is the set of primitive input bits on which a block's output bits depend. A disjoint decomposition ensures that the supports of two blocks, which do not have a predecessor-successor relationship, do not overlap; in other words, the DAG that results from the disjoint decomposition is a tree. Non-disjoint decompositions do not satisfy this property. Effective algorithms to compute the maximal disjoint decomposition of a circuit exist [4, 13].

Unfortunately, many circuits exist that cannot be described ef-

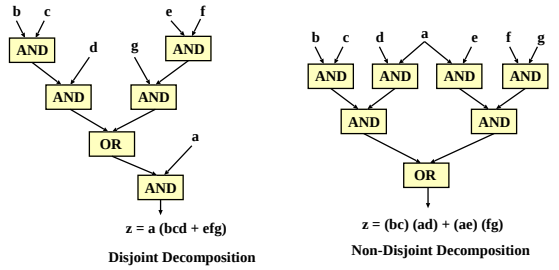

Figure 2: A disjoint decomposition can reduce the number of blocks, but may leave the circuit unbalanced compared to non-disjoint decompositions.

fectively in terms of fully disjoint decompositions. In some cases, disjoint decompositions can only find very large blocks, which provide a macroscopic, but not microscopic, structure on the circuit. The large blocks themselves are difficult to optimize, and exhaustive enumeration methods cannot optimize them within a reasonable running time.

Secondly, some circuits simply cannot be described in term of a fully disjoint decomposition, although, in some cases, their subcircuits can. Multipliers fall into this class of circuits. The disjoint decomposition of a multiplier is the multiplier itself, which yields no meaningful information in terms of structure and optimization. The first layer of an $m \times n$-bit multiplier is a partial product generator, i.e., an array of mn AND gates, and it expands the number of bits from $m + n$ to mn. No disjoint decomposition is possible within the partial product generator.

The remainder of the multiplier is a multi-input adder that sums the partial product bits, which we will refer to as a *compressor tree*, whose most favorable implementation uses carry-save arithmetic. Disjoint decompositions can be applied to compressor trees [19]. Unfortunately, the use of disjoint decompositions, in this context, requires some prior partitioning of the multiplier into the partial product generator and the compressor tree. If Boolean equations are provided for the entire multiplier, the logic synthesizer must first compute the partition prior to the decomposition.

Disjoint decompositions may also yield circuits that are unbalanced. As an example, consider the following Boolean expression:

$$z = abcd + aefg.$$

Fig. 2 shows the disjoint and non-disjoint decompositions of the circuit. The non-disjoint decomposition reduces the number of blocks, but leaves the circuit poorly balanced and increases the critical path delay. Although disjoint decompositions can optimize circuit area, they tend to increase the delay of the circuit in many cases.

For these reasons, many researchers have turned toward non-disjoint decompositions in recent years [5, 15, 23]. Unlike disjoint decompositions, maximal non-disjoint decompositions are not unique, and their performance is highly dependent on the partitioning of input bit into the support of different blocks.

Decomposition algorithms are either *top-down* or *bottom-up*. A top-down approach starts from a single block, i.e., the original circuit, and decomposes the block into two or more smaller blocks at each level; the latter approach starts from the input circuits, selects a subset of them to form a block, and then repeats the process, with the possibility of using the output bits of previously formed blocks as input bits to future blocks. In the case of disjoint decompositions, the input bits of a previously formed block are discarded, and new blocks can only be formed from the output bits of previously formed blocks; in contrast, non-disjoint decompositions may require that circuit inputs, and internal signals, be inputs to separate blocks. The performance of the top-down approach depends on the partitioning of the support set of a bigger block into smaller sets; the performance of the bottom-up approach depends on the choice of which input bits are selected as a set to form a block.

As the number of partitions of n input bits is exponential in n, it is intractable to exhaustively search the partition space at each level when performing a top-down decomposition. On the other hand, if k is a constant, the number of combinations of k bits from a set of n bits is polynomial in k. Thus, exhaustive enumeration of the partition space is feasible for a bottom-up decomposition, as long as k is sufficiently small and the method to evaluate the quality of the block formed from each combination is efficient.

Verma *et al.* have suggested an effective bottom-up decomposition named *Progressive Decomposition* [19], which computes a disjoint decomposition; however, Progressive Decomposition represents Boolean expressions using the *Reed-Muller* form. Although this representation is canonical, it is an inefficient representation for many arithmetic circuits. Specifically, some circuits exist for which the representation in *sum-of-product* or *product-of-sum* form may have a linear number of product term, whereas the *Reed-Muller* representation has an exponential number of product terms. Efficient algorithms, consequently, become inefficient for problem instances whose input sizes grow exponentially.

Progressive Decomposition also suffers from several theoretical weaknesses which we describe in Section 3, which render it inapplicable for circuits such as multipliers.

This paper presents a bottom-up non-disjoint decomposition that can be used to structure arithmetic circuits. Progressive Decomposition is a starting point, but its usage is relaxed in situations where non-disjoint decompositions are deemed to be preferable. The result is a less complex and more general approach. Additionally, our method does not use the *Reed-Muller* form for the circuit representation. Although our approach is applicable to any circuit, it is most useful for arithmetic circuits, due to the existence of numerous successful optimization techniques that are amenable to other classes of circuits. Although arithmetic circuits tend to be XOR-dominated, our approach is sufficiently robust to permit the inclusion of any kind of gate in the circuit.

The rest of the paper is organized as follows: In the next section we summarize related work on logic synthesis methods for arithmetic circuits. Section 3 summarizes Progressive Decomposition, which is the starting point of our algorithm. Sections 4 and Section 5 introduce our approach, which is then evaluated in Section 6. Finally, Section 7 concludes the paper.

2. STATE OF THE ART

A general approach to optimize arithmetic circuits is by using the high-level transformations such as those described by Verma *et al.* [20]. These transformations operate at the operation level, rather than the bit level; for example, basic operations include addition and multiplication, without specifying their implementation, e.g., a *ripple-carry* or *parallel-prefix* adder. These transformations are used to move disparate addition and multiplication operations that occur throughout a dataflow graph to form large multi-input adders, including the partial product reduction trees of the multipliers. These multi-input adders are then realized using compressor trees. Bit-level optimizations cannot be applied in this context, except for the specific methods used to generate the compressor trees [17].

Decomposition, introduced by Ashenhurst [2] and Curtis [8], have been a major focus in recent years. These techniques tend to be disjunctive, i.e., the Boolean functions are decomposed iteratively into block with disjoint support sets. Recent papers by

Bertacco *et al.* [4], Minato *et al.* [14] and Verma *et al.* [19] fall into this category.

Non-disjunctive decompositions of Boolean functions such as *Bi-decomposition* [15] and the *BDS* optimization system [23] split larger blocks into smaller ones, and their performance depends on the heuristics that balance the sizes of the decomposed blocks.

Progressive Decomposition [19] uses factoring methods based on symbolic algebra to decompose the functions, extracting a disjunctive decomposition of a Boolean function from its *Reed-Muller* representation. The limitations of Progressive Decomposition are enumerated in the following section, and the contribution of this paper is a set of techniques that mostly correct them. Despite its shortcomings, Progressive Decomposition yielded several novel results. Most notably, it is the first general logic synthesis technique to automatically convert a *ripple-carry* adder into a *carry-lookahead* adder; likewise, it could automatically infer the use of carry-save arithmetic when synthesizing compressor trees and parallel counters; and lastly, from a naive implementation, it automatically inferred the structure of a previously hand-optimized implementation of a leading zero detector [16], a component used in floating point arithmetic units. Thus, despite its shortcomings, which are emphasized and improved upon in this paper, Progressive Decomposition itself is quite powerful.

This paper presents a bottom-up, non-disjunctive decomposition that extends Progressive Decomposition. Our implementation uses compact *Binary Decision Diagrams (BDDs)* [1] instead of *Reed-Muller* form, but our ideas are generally amenable to any circuit representation that supports efficient algorithms for SAT testing and cofactor computation. The circuit in our algorithm is built progressively, but without the stringent stopping criteria that limits the effectiveness and generality of Progressive Decomposition; specifically, our method permits decomposed blocks to have overlapping support sets. Our method achieves comparable results to Progressive Decomposition on circuits where the latter has already established success, including adders and partial product reduction trees of parallel multipliers; moreover, it can optimize complete multipliers, and composite arithmetic circuits that have previously been implemented using sets of library components.

3. PROGRESSIVE DECOMPOSITION AND ITS SHORTCOMINGS

Fig. 3 illustrates the Progressive Decomposition algorithm [19]. Progressive Decomposition selects a subset B of input bits of a Boolean function, and produces a circuit, called a *block*, that condenses information about the input bits of B down to a smaller set of output bits B', called *Leader Expressions*. The complete circuit is then rewritten with the leader expressions replacing those in B. This naturally imposes a hierarchical structure on the circuit being optimized.

Progressive Decomposition ensures that the blocks that it forms satisfy the following two properties:

Information Condensation. The input-to-output ratio of each block is never below 1. This ensures that the replacement of B with the leader expressions computed by the block does not increase the number of inputs to the remainder of the circuit. The overall goal is to reduce the number of input bits at each layer as much as possible.

Disjunctive Decomposition. Any two blocks, which do not have a predecessor-successor relationship in the decomposed structure, must have disjoint support set.

Progressive Decomposition terminates when no further blocks satisfying these two properties can be found. Next, we describe the method to construct each block.

Choice of Input Bits: The correct choice of input bits to form a block is critical to achieving a well-optimized circuit. All combinations of k input bits, where k is a small constant, are considered; blocks are formed for each combination. The block that yields the smallest expression size, after rewriting the original expression in terms of leader expressions, is then chosen.

Leader Expressions Computation: Progressive Decomposition uses factoring based on symbolic algebra to compute leader expressions. The problem of merging leader expressions is reduced to the *Ideal Membership Problem*, which is solved using a known algorithm that runs in exponential worst-case time [3].

Minimization of Leader Expressions: Redundant leader expressions are removed, and relations among the leader expressions are computed for use in later steps.

Rewriting the Input Expression: New variables are introduced for each leader expression; the original expression is rewritten in terms of the new variables.

Progressive Decomposition suffers from several drawbacks. The first is that it is based on the *Reed-Muller* form, which can be exponentially large compared to other compact representations, such as factored form [6], for many circuits of interest.

Second, the algorithm to select input bits to optimize *progressively* is complex and computationally intensive, which restricts the use of this technique to small circuits. The selection of input bits with which to form blocks is exhaustive; for each set of bits considered, the *Ideal Membership Problem* must be solved several times, often for large Boolean expressions. The criteria for selecting the best combination of input bits is not completely sound, as it measures the size of the rewritten Boolean expression, which may not be an exact indicator of the circuit's overall delay.

Often, when Progressive Decomposition fails to reduce the number of input bits at each step, it simply stops. As an example, consider two-bit multiplication, i.e., $\langle a_1 a_0 \rangle \times \langle b_1 b_0 \rangle$. Assume that the leader expressions of input bits a_0 and b_0 are to be computed. Since the second output bit of the multiplier is $(a_0 b_1 \oplus a_1 b_0)$, the leader expressions of a_0 and b_0 must contain all information about a_0 and b_0 which is required to compute this expression. For $a_1 = 0, b_1 = 1$ the value of the expression is a_0, and for $a_1 = 1, b_1 = 0$ the value of the expression is b_0. This means the leader expressions of a_0 and b_0 must be able to generate both a_0 and b_0, which forces a_0 and b_0 to be leader expressions. Thus, the leader expressions are the same as the initial set of inputs. Depending on its implementation, Progressive Decomposition would either stop, assuming that the circuit has been optimized as far as possible, or it would become stuck in an infinite recursive loop. In the case of a multiplier, the leader expressions of any combination of input bits are always the same as the original set of bits. The method proposed in this paper is designed to handle such cases efficiently by allowing non-disjoint decompositions.

Progressive Decomposition, it should be noted, can optimize the partial product reduction tree of a multiplier, but it cannot optimize the complete multiplier, as described above. For an $m \times n$-bit multiplier, the partial product generator expands $m + n$ input bits into $m \times n$ partial product bits. Progressive Decomposition, by design, always tries to condense the input bits into fewer expressions, so it generally performs poorly for circuits with high fanout.

In fact this is a key problem of all disjoint decomposition techniques. Similar phenomena, furthermore, tend to occur in many composite arithmetic circuits. For example, Progressive Decomposition can optimize the circuit in Fig. 1 approximately halfway, but cannot produce the best implementation shown in Fig. 8.

To summarize, input condensation and disjunctive decomposition are overly restrictive, and limit the effectiveness of Progressive Decomposition; they must be relaxed if Progressive Decomposition is to be extended to handle multipliers and composite arithmetic circuits.

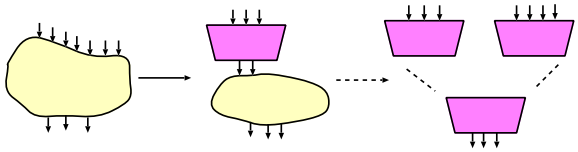

Figure 3: Progressive decomposition iteratively builds larger blocks from smaller ones, with each block having a input-to-output ratio of at least one. Between any two blocks, either the support set of one block is completely contained in the support set of the other, or the support sets of the two blocks are disjoint.

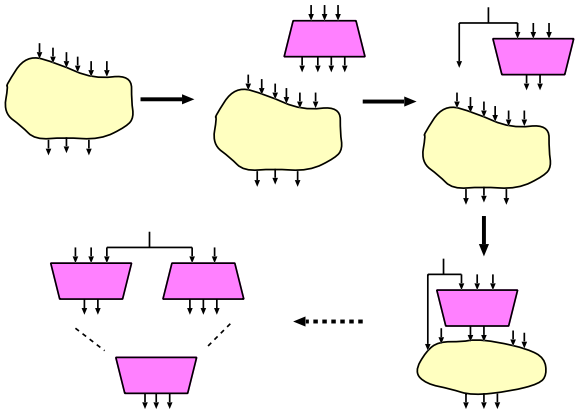

Figure 4: The algorithm described in this paper permits blocks with overlapping support sets between blocks that do not have a predecessor-successor relationship, and permits the formation of blocks whose input-to-output ratio is less than one.

4. MAIN IDEA

This section presents a new approach that overcomes the major drawbacks of Progressive Decomposition. In principle, any input representation can be used, as long as a minimization algorithm for that representation exists. Our implementation uses BDDs and requires efficient methods to solve *SAT* and to compute the *Restrict* operator for the generalized cofactors. We use the algorithm suggested by Coudert and Madre [7] to implement this operator.

In our approach, most leader expressions are computed using random sampling as discussed in Section 5.2; the remaining ones are computed by checking the functional dependency of the original expression on the set of leader expressions: as discussed by Lee *et al.* [10], the problem at hand is reduced to an instance of SAT. Although SAT is NP-complete, the solver is called once to compute the set of leader expressions of one group, unlike Progressive Decomposition, which solves the *Ideal Membership Problem* repeatedly.

To select the appropriate input bits for each block, we use a heuristic based on the delay estimator of a circuit proposed by Macii *et al.* [12], rather than running a detailed timing analysis algorithm for each combination of input bits. Given a small constant, k, we consider all combination of k input bits. The delay estimator suggests which input bits are likely to occur on the critical path, and those bits are chosen as the inputs of the next block; this approach is much more accurate than using the size of a function's *Reed-Muller* representation as a proxy for circuit delay.

Leader expressions that do not help to reduce the circuit delay can be discarded. One such example is a leader expression that is simply an input bit. The resulting decomposition may not be

```
// The function takes the list of input expressions,
// and creates an architectural implementation using
// a bottom-up approach.
decomposeCircuit (List ℒ) {
do {
    // This step uses a simple delay estimator to choose
    // the right combination, which leads to the smallest
    // delay, when chosen as the input of a block.
    combinations = generatePrunedSet (ℒ, k);
    rightComb = chooseComb(combinations, delayEstimator);

    // Here we decide (1) whether a leader expression is
    // a real leader expression, and (2) disjunctive or
    // non-disjunctive decomposition should be performed.
    LE = generateLEsViaAssignments(ℒ, rightComb);
    LE = LE ∪ generateMissingLEs (ℒ, rightComb, LE);
    LE = pruneLEs (ℒ, LE);

    ℒ = rewriteExpr(ℒ, LE); }
  while(all elements in ℒ are not literals); }
```

Figure 5: An overall description of our algorithm.

disjoint, i.e., distinct blocks may have overlapping support sets. Our algorithm does not minimize the number of leader expressions, which permits the input-to-output ratio to be less than one. The relaxation of this restriction is critical to permit our method to handle complete multipliers.

The *Restrict* operator computes the generalized cofactor of one Boolean expression with respect to another, and replaces occurrences of leader expressions by new variables in the non-disjoint decomposition. For example, let F be the original expression, and g be a leader expression. The Shannon expansion of F with respect to g is

$$F = g(E \mid g) + \overline{g}(E \mid \overline{g}),$$

where $(E \mid g)$ and $(E \mid \overline{g})$ are the generalized cofactors of E with respect to the leader expression g. Since the generalized cofactors are not unique, a different cofactor may give a different result. In the future, we hope to extend the method presented in this paper to sidestep the computation of generalized cofactors. If F is sufficiently small, we convert it to *Reed-Muller* form, and rewrite it using the leader expressions, as in Progressive Decomposition. Fig. 4 illustrates the algorithm and emphasizes the points where it differs from Progressive Decomposition.

5. OUR APPROACH

This section describes our algorithm in detail; Fig. 5 shows the various subroutines used in our algorithm. The following subsections summarize each of these subroutines in detail.

5.1 Choosing the Input Bit Combination

The performance of a bottom-up decomposition depends on the chosen group of input bits to each block. A collection is a set of sets; at each layer, there will exist several blocks corresponding to several sets of input bits. A *collection* is a set of whose elements are the k-bit sets of input bits that have been chosen, where k is a small constant. Ideally, all combinations of k input bits would be chosen and all their collections should be considered as the inputs of blocks at a level. In other words, the inputs of block at a level should be chosen at once. However, this would result in an exhaustive exponential search. In contrast, we use a greedy heuristic that progressively determines the best group of input bits for a block at each stage. Note that the heuristic is well suited for arithmetic circuits, which usually have effective online algorithms; this means that the information of some of the bits can be encoded into fewer bits without affecting the other bits.

The first step is to choose a combination of k input bits to form

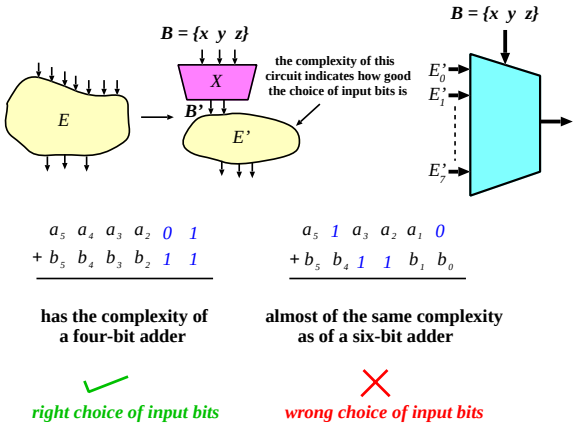

Figure 6: An example illustrating how to decide whether a choice of input bits is right.

leader expressions. All combinations are considered. For each combination of bits, we estimate the impact on the complexity of the expression when these bits form a block; the subset that has the highest impact on complexity is chosen.

To estimate the impact on complexity, our algorithm constructs a block X for the chosen set of input bits B. Fig. 6 shows the reduced expression E' after choosing a set of input bits and replacing their leader expressions by new variables in the original expression, E. The complexity of X must be determined, along with the complexity of the reduced expression, E'. Contrary to Progressive Decomposition, our algorithm considers various assignments of values to the chosen bits, and computes the original expression given these assignments. Since the number of chosen bits is small, at most 6 in our experiments, this is not computationally intensive. The average complexity of the reduced expression E is a reasonable measure of the complexity of E'.

For example, consider the adder shown in Fig. 6, and suppose that the four bits corresponding to the two least significant bits of both integers are chosen. For each assignment of these bits, the complexity of the circuit is the same as the complexity of a four-bit adder; however, if a different set of four bits are chosen, e.g., the right-hand side of Fig. 6, then the complexity of the circuit is estimated to be approximately the same as that of a six-bit adder. It follows that the first choice is better than the second.

We use a method described by Macii *et al.* [12], to estimate the delay as a function of number of variables, size, entropy and density (i.e., the difference between the expected value of the expression and 0.5). This method is imprecise, but effectively computes the relative delays of two circuits in comparison with one another; this is precisely what we need.

The input bits may not have the same delay; thus, we must also consider the complexity of the block X that has been formed. Our algorithm models the circuit as a multiplexer whose control bits are the chosen bits and whose input values are the E'_i, i.e., the reduced expression E under various assignments of the chosen bits. Next, we estimate the delays of E'_is and of the chosen bits, and based on that estimate, the delay of the multiplexer. The combination for which this delay turns out to be minimum is selected.

5.2 Computation of Leader Expressions

Let E be the input expression, B be the set of bits for whom we wish to compute leader expressions, and R be the remaining bits. One possibility, which runs in exponential time, is to enumerate all possible assignments of variables in R, compute E using these values, and store the resulting expressions for each assignment in a set S. S, effectively, contains the leader expressions, because all information regarding the bits in B is contained in S. The final step is to remove all of the redundant leader expressions from S, including those that are dependent on others. The problem with this approach, as stated above, is its exponential time complexity.

Instead, we randomly sample subsets of the complete assignment space of R, knowing that random sampling may miss some leader expressions. For example, the following expression computes the carry output of a four-bit adder:

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

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 leader expression. The probability of missing the expression, furthermore, increases exponentially with the bitwidth of the input integers.

Instead, we perform random sampling but we include an analytical model that helps us to find missing leader expressions; this problem is reduced to that of determining functional dependencies, which, in turn, is modeled using SAT [10]. Let S be the set of leader expressions found using random sampling; we want to determine if all of the required information about the input bits B is contained in S. This is effectively the same as deciding whether using the values of expressions in S and the value of bits in R one can uniquely determine the value of the original expression E; in other words, the task is to determine 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 E can be written as a function of the expressions in S; if not, then S is missing some leader expressions. Suppose, for example, that two assignments of variables exist for which all expressions in S have the same value, but E has a different value; then E is not a function of the expressions in S and more information is needed to uniquely compute E. The corresponding assignment of variables in R will generate the missing leader expressions.

Lee *et al.* [10] provide an approach based on SAT to determine the functional dependency. 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 a set 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:.

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

Here t is the number of leader expressions in S. Satisfying assignments provide the missing leader expressions, and S contains all of the leader expressions if and only if there are no satisfying assignments.

This application of SAT does not require sampling; however, omitting sampling may lead to multiple satisfying assignments that lead to the same leader expression. We use random sampling to reduce the number of satisfying assignments in the SAT instance to a small number. In the sampling phase of our experiments, we choose $10,000$ random assignments of variables, which yields more that 90% of the leader expressions.

5.3 Pruning the Set of Leader Expressions

Here, we remove redundant leader expressions and determine

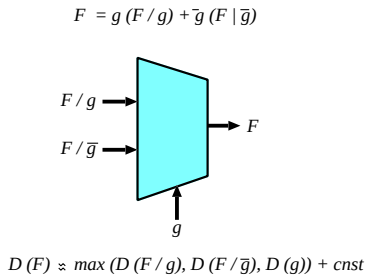

Figure 7: Shannon decomposition with respect to generalized cofactors.

whether a non-disjunctive decomposition is necessary. We also minimize the set of leader expressions when doing so helps to reduce the complexity of the circuit.

If a leader expression is the same as an input bit, then a non-disjunctive decomposition is required. This is because that particular bit is needed somewhere else in the circuit in its raw form. All such leader expressions are removed from the set.

In some cases, enumerating the assignments of remaining bits produces leader expressions that contain no information that helps to compute E. For example, consider the expression $(a_0b_1+a_1b_0)$, and let us compute leader expressions of a_0 and b_0. The assignment of $a_1 = b_1 = 1$ yields $(a_0 + b_0)$ as a leader expression, which cannot be used effectively to compute the original expression; this type of useless leader expression can be safely discarded.

To check for these cases, we compute the generalized cofactor of the original expression with respect to the underlying leader expression and its complement. If any of the two cofactors are at least as complex as the original, we deduce that the underlying leader expression contains no meaningful information that can help us to compute the original expression. Consider the following example:

$$\begin{aligned} E &= (a+b)x + (ab+c)y, \\ g &= a+b+c, \\ E \mid \overline{g} &= 0, \\ E \mid g &= (a+b)x + (ab+c)y. \end{aligned}$$

Since $E \mid g = E$, we can safely discard g.

The removal of redundant leader expressions uses a reduction to a SAT instance, as in the preceding section. We wish to determine whether a particular leader expression can be written as a function of the others. Since the number of variables here is small, this SAT instance will be small as well.

5.4 Rewriting the Expression Using Leader Expressions

The final step is to rewrite the original expression in terms of the leader expressions. This is done as the Shannon expansion of the original expression with respect to the leader expressions using the *Restrict* operator. We use the implementation of *Restrict* operator given by Coudert and Madre [7]. Since generalized cofactors are not unique, the rewritten expression may depend on the order in which the cofactors of each leader expression are computed. We estimate the delays of the cofactors with respect to each leader expression; based these delays, we then estimate the delay of the original expression, implemented using a Shannon expansion, as illustrated in Fig. 7. The leader expression for which the delay of the Shannon expansion is minimal is selected first during the rewriting process.

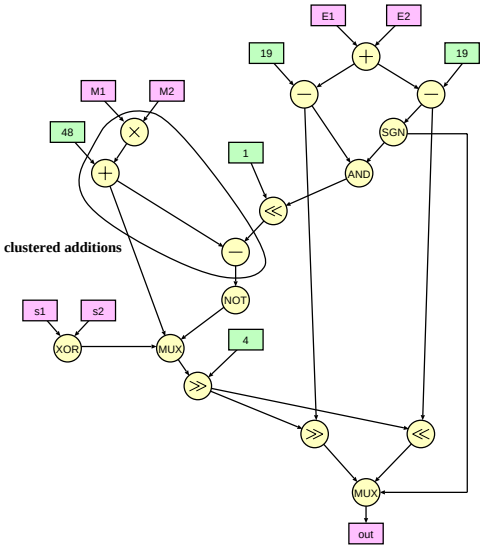

Figure 8: The circuit shown in Fig. 1 can be transformed using arithmetic identities in such a way that the additions and subtractions can be clustered together resulting in a larger compressor tree.

Small input expression are converted to *Reed-Muller* form and are rewritten, as done by Progressive Decomposition. The motivation to rewrite the original expressions in terms of leader expressions is to help find the next group of leader expressions; if we could modify our algorithm to compute the leader expressions in such a way that only the leader expressions found so far are required, then this rewriting step would become unnecessary; however, we have not yet found an efficient way to do so.

6. EXPERIMENTAL RESULTS

We have implemented our algorithm in C++ using the BDD package *BuDDy* [11]. Our tool takes the input expressions in a simplified blif format featuring four logic gates, (AND, OR, NOT, XOR). The user may also provide information regarding the place of an input bit within an integer. The tool outputs a VHDL description of the optimized circuit. We compare the circuits produced by our algorithm to Progressive Decomposition and library/manually optimized versions of each circuit. All the circuits are synthesized using a common standard cell library for UMC $90nm$ CMOS technology, with the optimization strategy to minimize delay.

We have divided our benchmarks into two categories: basic arithmetic components and composite arithmetic circuits. Table 1 shows the results of our experiments compared to a baseline circuit description. The multiplier (a component) and the composite circuits show that our method is more general than Progressive Decomposition in the circuits that it can optimize; meanwhile, the remaining three components, an adder, ternary adder, and barrel shifter, show that our approach achieve comparable results to Progressive Decomposition for circuits where the latter is able to terminate. Progressive Decomposition effectively optimizes the delay of the ternary adder, but increases the delay of the adder and the barrel shifter; it does, however, effectively reduce the area of the adder.

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*; the manually optimized circuit is taken from a well-established component library, and appears to be a prefix adder. The logic

Benchmark	Original		Progressive Decomposition		**Our Algorithm**		Library / Manual Optimization	
	Delay (ns)	Area (μm^2)	Delay (ns)	Area (μm^2)	**Delay (ns)**	**Area (μm^2)**	Delay (ns)	Area (μm^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 **(– 20%)**	0.57 (– 38%)	6200 (– 18%)
16-bit, 4-stage Barrel Shifter	0.14	5408	0.16 (+ 14%)	5504 (+ 2%)	0.14 **(0%)**	5395 **(0%)**	0.13 (– 14%)	5900 (+ 9%)
adpcmcoder [9]	0.85	5678	–	–	0.46 **(– 46%)**	5121 **(– 10%)**	0.49 (– 42%)	4901 (– 14%)
G721 [9] (fmult)	0.94	5142	–	–	0.82 **(– 12%)**	7998 **(+ 55%)**	0.83 (– 11%)	7113 (+ 38%)
Sum-of-Absolute Differences (SAD) [18]	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 decompose the circuit.

synthesizer is also able to improve the original circuit architecture using algebraic factorization.

The second component is a 12-bit ternary adder built from two *Ripple Carry Adders* conjoined in sequence. Due to the mixture of XOR and 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 our algorithm and Progressive Decomposition transform the first adder into a carry-save adder to compute the sum more efficiently. The library design is slightly faster than the ones produced by our algorithm and Progressive Decomposition due to small variations in the design of the final adder, i.e., a hybrid adder is used in the library design.

The third component is an 8×8-bit shift-and-add multiplier. The logic synthesizer is unable to optimize the original circuit. Progressive Decomposition was unable to process this circuit because of exorbitantly large input expressions in *Reed-Muller* form. Even if the size was not a problem, Progressive Decomposition would not have been able to optimize it due to the reasons outlined earlier in the paper. Our algorithm, in contrast, produced a circuit that was marginally faster and smaller than the library implementation. Our algorithm is able to exploit the correlation among the partial product bits, which apparently were ignored by the library implementation.

The fourth component is a 16-bit Barrel-shifter. Although the circuit is not arithmetic in nature, it is considered because of high fan-in dependency between inputs and outputs. Progressive Decomposition was unable to find any block with an input-to-output ratio of less than one, and was therefore unable to optimize it. Additionally, the conversion to the XOR-dominated *Reed-Muller* form increases the delay compared to a more traditional implementation based on multiplexers. Since the synthesis tools can already optimize this type of circuit using algebraic factorization, the delays and areas of the three implementations other than Progressive Decomposition were comparable.

Next, we turn our attention to the three compound circuits. The first compound circuit comes from the adpcmcoder [9] benchmark and it computes $\frac{(\mathsf{delta}+0.5)\times \mathsf{step}}{4}$, where delta is a 3-bit integer and step is a 16-bit integer. Progressive Decomposition could not handle the large input expression. The manual implementation was produced by the transformations of Verma and Ienne [21], and was significantly faster than the original; four distinct addition operations were clustered together to form a 4-input adder at the bottom of the circuit; the compressor tree that implements this multi-input adder was generated using the 3-greedy algorithm of Stelling *et al.* [17]. Our approach, in contrast, has no prior knowledge that the desired circuit is a compressor tree, and therefore, is much more general than the methods used to produce the manually optimized version. Altogether, the result produced by our method is slightly faster and slightly larger than the manual design.

The second composite circuit is shown in Fig. 1. Progressive Decomposition partially optimizes the circuit, but leaves a large expression in *Reed-Muller* form that could not be synthesized. Our algorithm, on the other hand, found an implementation which is 12% faster than the original implementation. Since, the circuit is not a standard component, there are no library implementations available; we manually tried to improve the delay of the circuit by clustering some separated additions using algebraic identities. The result was marginally slower (0.01 ns) than the method produced by our algorithm, but significantly smaller. It is important to note that the logic synthesis method described in this paper only optimizes for delay, and does not give any consideration to area when making decisions.

The third composite circuit is the *Sum-of-Absolute Differences (SAD)* computation, which is used in a variety of video coding algorithms such as H.264. The manually optimized implementation was suggested by Vassiliadis *et al.* [18], and uses a slower implementation of absolute difference, but it helps in clustering additions. Since in the chosen circuit there are only two absolute differences are being added, the penalty for having a slower absolute difference did not pay off. Our approach, however produces a circuit which is almost 18% faster than the naive implementation. Progressive Decomposition was unable to optimize this benchmark due to its high computational complexity.

The primary goal of the optimization method presented in this paper is to effectively improve naive implementations of a circuit so that they resemble optimized implementations, thus, alleviating the need for the designer to perform time-consuming manual optimizations by hand; our goal is not to improve the quality of circuits that have already been optimized by hand. Thus, we care more about the ability of our technique to automatically discover good circuit designs, such as transforming a *Ripple Carry* into an *Carry Lookahead* adder, rather than optimizing the latter, for example, by adjusting the group size.

The runtime of our approach, as implemented here, depends on the size of the BDD. For the circuits listed in Table 1, the longest running benchmark was adpcmcoder, which required approximately 28 minutes. Although it optimized an 8×8-bit multiplier, it failed to terminate for a 16×16-bit multiplier. Consequently, some scalability issues must still be resolved before the proposed technique becomes useful in practice; we intend to address these issues in the future.

7. CONCLUSIONS AND FUTURE WORK

This paper has introduced a new method for logic synthesis of arithmetic circuits. It addresses many of the shortcomings of Progressive Decomposition by using a non-disjunctive decomposition

method instead, while building larger blocks from smaller ones progressively. It sidesteps the theoretical weaknesses and computational complexities that plague Progressive Decomposition's use in practice. For an adder, ternary adder, and barrel shifter, our method achieved comparable results to Progressive Decomposition; it was also able to optimize a multiplier and three composite circuits for which Progressive Decomposition was ineffective. Although the algorithm proposed here is based on BDDs, it can easily adapt to any circuit representation that has efficient algorithms for minimization, SAT, and generalized cofactor computation. At present, the size of the BBDs that we use limits the scalability of the proposed approach in the case of large-bitwidth multipliers. We hope to overcome this limitation in the future through the use of a more compact representation and by employing techniques with faster running times for each of the main steps in the algorithm.

8. REFERENCES

[1] S. B. Akers. Functional testing with binary decision diagrams. In *Proceedings of the 8th International Conference on Fault-Tolerant Computing*, pages 75–82, Toulouse, France, June 1978.

[2] 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.

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

[4] 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.

[5] D. Bochmann, F. Dresig, and B. Steinbach. A new decomposition method for multilevel circuit design. In *Proceedings of the conference on European design automation*, pages 374–77, Amsterdam, Netherlands, June 1991.

[6] R. K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–98, Mar. 1987.

[7] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 126–29, Santa Clara, Calif., 1990.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] Y. Matsunaga. An exact and efficient algorithm for disjunctive decomposition. In *Proceedings of the 8th Workshop on Synthesis and System Integration of Mixed Technologies*, pages 44–50, Sendai, japan, Oct. 1998.

[14] 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.

[15] 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.

[16] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-2(1):124–28, Mar. 1994.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.