Timing-Driven Logic Bi-Decomposition

Jordi Cortadella, *Member, IEEE*

***Abstract*—An approach for logic decomposition that produces circuits with reduced logic depth is presented. It combines two strategies: logic bi-decomposition of Boolean functions and tree-height reduction of Boolean expressions. It is a technology-independent approach that enables one to find tree-like expressions with smaller depths than the ones obtained by state-of-the-art techniques. The approach can also be combined with technology mapping techniques aiming at timing optimization. Experimental results show that new points in the area/delay space can be explored, with tangible delay improvements when compared to existing techniques.**

***Index Terms*—Bi-decomposition, delay optimization, logic decomposition, tree-height reduction.**

I. INTRODUCTION

DELAY optimization can be tackled at different stages of circuit synthesis, from high-level to layout. This paper focuses on technology-independent logic synthesis techniques for combinational circuits [1] that typically precede technology mapping.

Given the complexity of the problem, delay optimization is usually performed after the size of the Boolean network representing the circuit has been reduced. Numerous multilevel logic synthesis techniques exist for that, either by using algebraic [2], [3] or Boolean methods [4]. Most of the techniques on technology-independent delay optimization aims at reducing the depth of Boolean networks by restructuring [5]–[7]. Even the depth of a network is not an accurate estimation of the circuit delay; both have a high correlation. For this reason, the depth of the network is a parameter frequently used in technology-independent optimization techniques.

Reducing the size of a Boolean network often implies the extraction of common subexpressions that can be shared in several subnetworks. As a side effect, sharing may also lead to increasing the depth of the network. Thus, when delay is the parameter under optimization, sharing logic is not always a good approach for logic decomposition. Even if we disregard the delays produced by the fanout capacitances, increasing the degree of sharing may negatively affect the performance of a circuit.

Fig. 1 depicts three different circuits implementing the same Boolean function. Each circuit is represented by a directed acyclic graph (DAG) of two-input gates. The bubbles on the arcs represent inverters. The depth of the circuit is calculated as

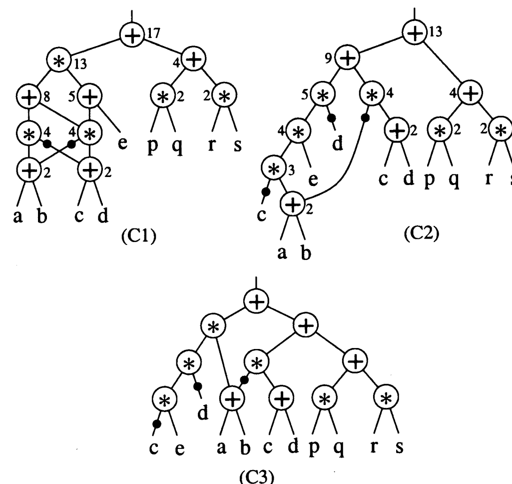

Fig. 1. Three different circuits implementing the same Boolean function.

the number of nodes of the longest path in the DAG (inverters are ignored). A DAG can be unfolded in such a way that no multiple-fanout nodes exist, except for the inputs of the circuit, thus obtaining a tree with the same depth. The numbers annotated to each node indicate the number of paths crossing the node that corresponds to the number of leaves of the tree version of the DAG. A lower bound on the depth of a DAG G is $\lceil \log_2 p \rceil$, p being the number of paths of G, and assuming that it can only be transformed by rules that cannot reduce the number of nodes [8].

Even though C1 and C2 have the same number of nodes and C2 has more levels than C1, the lower bound on their depth is different due to their sharing degree. Given that the tree version of C1 has 17 leaves, a lower bound on the depth of C1 is five levels. Therefore, any restructuring of C1 that does not reduce the number of nodes, will never achieve a depth smaller than five. On the other hand, circuit C2 offers more chances for optimization, since it has only 13 paths and the lower bound on its depth is four levels. Circuit C3 depicts a possible restructuring of C2, by applying the associative law, that reduces its depth. This example shows that executing an aggressive area-oriented algorithm may prevent one from obtaining the desired number of logic levels during timing optimization.

In [9], a technique that performs logic decomposition during technology mapping is proposed. With this approach, the inaccuracies introduced by splitting these two phases are reduced at the expense of a high computational cost. More details on this technique are discussed in Section VII-B.

This paper proposes a different approach that simultaneously combines functional decomposition [10] and delay optimization. In particular, it combines two strategies: tree-based bi-decomposition of Boolean functions and tree-height reduction of Boolean expressions.

Manuscript received September 22, 2002; revised December 27, 2002. This work was supported in part by a grant from the Intel Corporation and by a grant from CICYT TIC2001 2476. This paper was recommended by Guest Editor L. Stok.

J. Cortadella is with the Department of Software, Universitat Politècnica de Catatunya, Barcelona 08034, Spain (e-mail: jordi.cortadella@upc.es).

Digital Object Identifier 10.1109/TCAD.2003.811447

0278-0070/03$17.00 © 2003 IEEE

The approach aims at finding the minimum-depth tree for a Boolean function. It builds the tree from root to leaves by using bi-decomposition techniques [11], [12], and reduces the depth by means of rewrite rules that apply the associative, commutative and distributive laws of the Boolean algebra.

The relevance of field programmable gate arrays (FPGAs) based on look-up tables (LUTs) in the last decade has fostered various efforts in finding effective methods to decomposed functions [13]–[15]. Since each LUT is able to realize any arbitrary function up to a certain number of inputs, these methods are mostly oriented to partition the support of the components. Our goal, however, is to find efficient decompositions for cell-based designs in which the functionality of each component is relevant.

The main contributions of the technique presented in this paper are the following.

- Bi-decomposition and depth reduction are interleaved during the global decomposition of a function (Section V). The existing approaches perform both functions as clearly separated steps.
- A heuristic search for the application of transformations for tree-height reduction is proposed (Section IV-C).
- A new strategy for bi-decomposition based on function approximations is proposed. This technique subsumes previous existing approaches based on decompositions of binary decision diagrams (BDDs). Moreover, algebraic factorization is also used as an alternative method for bi-decomposition (Section V-A).

The paper is organized as follows. Section II gives an overview of the approach and is illustrated with an example. Section III introduces the representation of binary DAGs and the rewrite rules. Section IV proposes algorithms for an efficient exploration of the transformations for tree-height reduction. Section V presents the main algorithm for logic decomposition. Experimental results are reported in Section VI. Finally, Section VII discusses related work.

II. Overview

This section illustrates the main paradigm of the approach for logic decomposition. First, some background on tree-height reduction is presented. Next, a step in the recursive progress of the main algorithm is described with an example.

A. Tree-Height Reduction: An Example

Tree-height reduction [16] was originally proposed in the scope of optimizing compilers for the generation of code in multiprocessor systems. Fig. 2. illustrates an example. The tree in Fig. 2(a) represents a factored form for the Boolean expression

$$ab + acd + acef + acegh. \quad (1)$$

If we assume zero arrival time for all inputs and unit area ($a = 1$) and unit delay ($d = 1$) for each node, the tree is characterized by the pair ($a = 7$, $d = 7$).

The tree in Fig. 2(b) is the one obtained by SIS after executing the speed_up command [6]. This tree is characterized by the pair ($a = 9$, $d = 5$). A more efficient implementation can be

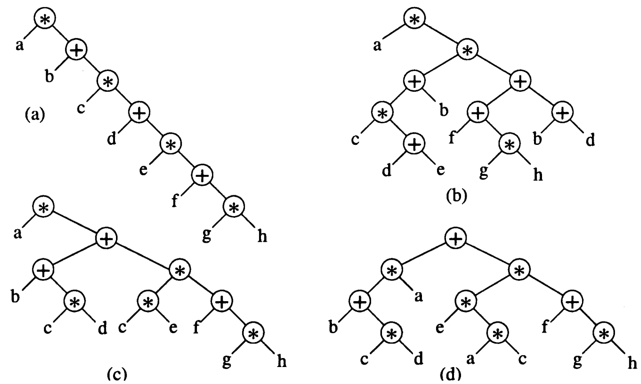

Fig. 2. Equivalent factored forms.

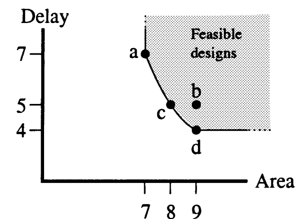

Fig. 3. Area/delay tradeoff for the trees.

found by applying simple transformations (associative and distributive laws) to the original tree. It is shown in Fig. 2(c) with ($a = 8$, $d = 5$). Finally, by further applying transformations, the tree in Fig. 2(d) can be obtained with ($a = 9$, $d = 4$). It would not be difficult to prove, for this particular example, that the solutions shown in Fig. 2(a) and (d) are optimal in area and delay, respectively. The tree obtained by speed_up is suboptimal, since there are other equivalent trees with the same area and shorter delay [Fig. 2(d)] or the same delay and smaller area [Fig. 2(c)].

Fig. 3 shows a diagram representing the space of feasible designs for expression (1). The points (7,7), (8,5), and (9,4) are optimal in the sense that there is no other design that can improve area and delay. However, the point (9,5) obtained by the speed_up command is suboptimal.

B. Algorithm

Fig. 4 depicts an example of the approach presented in this work. The boxes represent sums of products in matrix form (each row is a term). The main algorithm uses recursion to decompose a Boolean function from root to leaves. Each call in the recursion tree consists of the following steps.

1) The Boolean function is decomposed into two subfunctions and a Boolean operator (bi-decomposition). The methods for bi-decomposition are discussed in Section V-A.
2) The two subfunctions are decomposed into a binary tree by a fast algebraic factorization algorithm [17].
3) The binary tree is heuristically balanced by using tree-height reduction transformations. In the figure, the shadowed nodes indicate the points where the distributive law is applied. The tree is further balanced by applying the associative law. The algorithms for tree-height reduction are presented in Section IV.

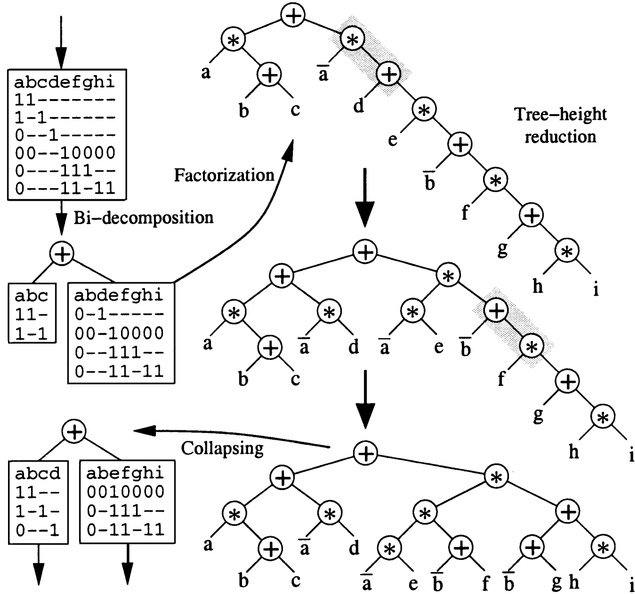

Fig. 4. Example of timing-driven bi-decomposition.

4) The left and right children of the tree are collapsed and the process is recursively repeated for each child.

Step 1 uses the full power of Boolean algebra for decomposition. Steps 2 and 3 are algebraic. For this reason, Step 4 collapses subtrees in such a way that Boolean decomposition is applied at each node of the tree.

III. Binary DAGs and Trees

Single-output circuits are represented by rooted DAGs. Each internal node has two children and is labeled with a Boolean operator (AND or OR). Leaf nodes are labeled with (possibly complemented) literals. Henceforth, we will assume that all DAGs are reduced, i.e., they do not have more than one instance of isomorphic sub-DAGs under the application of commutativity to the children. In case they are not reduced, isomorphic copies of sub-DAGs can be removed by keeping only one of them and changing the arcs accordingly. This transformation must be iteratively applied until no more isomorphic sub-DAGs appear. Henceforth, we will call binary DAGs (BDAGs) the DAGs representing circuits as described above.

A BDAG can be unfolded and uniquely represented by a binary tree (see Fig. 1). This tree is called the tree version of a BDAG (denoted by G^{Δ}). Similarly, a tree can be uniquely represented by a BDAG by sharing all isomorphic subtrees. Given that BDAGs are reduced, there is a one-to-one correspondence between BDAGs and binary trees.

Given a binary tree T, we will refer to T as the root node or the tree itself. The following nomenclature will be used for binary trees:

T.left, T.right	Left and right children
CHILDREN(T)	$=\{T$.left, T.right$\}$
T.op	Type of node: $+$, $*$, or $\perp$(literal)
$\lvert T.\rvert$	Number of leaves of the tree
$\mathcal{D}(T)$	Depth of the tree.

We can also represent trees as triples

$$T \equiv (T.\text{op}\ T.\text{left}\ T.\text{right}).$$

Note that, for binary trees, $|T|$ is equivalent to the number of nodes of the tree plus one. The depth of a tree is defined as follows:

$$\mathcal{D}(T) = \begin{cases} 0, & \text{if } T.\text{op} = \perp \\ 1 + \max(\mathcal{D}(T.\text{left}), \mathcal{D}(T.\text{right})), & \text{otherwise} \end{cases}.$$

The definitions above can easily be extended to BDAGs. The number of leaves of a tree is analogous to the number of paths of a BDAG. The number of paths of a BDAG G, denoted by $\Pi(G)$ is defined as follows:

$$\Pi(G) = \begin{cases} 1, & \text{if } G.\text{op} = \perp \\ \Pi(G.\text{left}) + \Pi(G.\text{right}), & \text{otherwise} \end{cases}.$$

Theorem 1:

$$\Pi(G) = |G^{\Delta}|.$$

Proof: Obvious from the one-to-one correspondence between BDAGs and binary trees.

A. ACD Rewrite Rules

Trees and BDAGs can be transformed by using the commutative (C), associative (A), and distributive (D) laws of Boolean algebra (ACD-rules)

$$\begin{aligned} \text{A}:\quad & (+\,T_1\,(+\,T_2\,T_3)) \equiv (+\,(+\,T_1\,T_2)\,T_3) \\ & (*\,T_1\,(*\,T_2\,T_3)) \equiv (*\,(*\,T_1\,T_2)\,T_3) \\ \text{C}:\quad & (+\,T_1\,T_2) \equiv (+\,T_2\,T_1) \\ & (*\,T_1\,T_2) \equiv (*\,T_2\,T_1) \\ \text{D}:\quad & (+\,T_1\,(+\,T_2\,T_3)) \equiv (*\,(+\,T_1\,T_2)(+\,T_1\,T_3)) \\ & (*\,T_1\,(+\,T_2\,T_3)) \equiv (+\,(*\,T_1\,T_2)\,(*\,T_1\,T_3)). \end{aligned}$$

One of the main subproblems in this work is the exploration of different BDAG representations for Boolean expressions. This exploration is done under the assumption that a minimum-size BDAG is given (e.g., point a in Fig. 3). By iteratively applying transformations, different solutions are obtained. These solutions draw the curve determined by the optimal solutions with regard to the area/delay tradeoff.

In order to have a monotonic behavior of the exploration, the D rule is only applied from left to right, i.e., no transformations extracting common factors will be used. Although this strategy impedes a wider exploration of BDAGs, it guarantees termination and works reasonably for multilevel netlists whose size has been reduced by an iterative extraction of divisors.

A side effect of using the D-rule from left to right is that the number of paths of the transformed graph is never reduced. Hence, the following theorem holds.

Theorem 2 (Lower bound on depth): Let G' be a BDAG obtained from G by applying the ACD-rules. A lower bound for $\mathcal{D}(G')$ is

$$\mathcal{D}(G') \geq \lceil \log_2 \Pi(G) \rceil. \qquad (2)$$

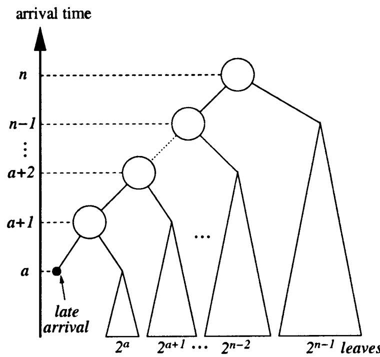

Fig. 5. Illustration of the proof of theorem 3.

Proof: Given that the ACD-rules never reduce the number of paths, we have that $\Pi(G') \geq \Pi(G)$. By Theorem 1, we also know that $\Pi(G) = |G^{\Delta}|$. The theorem immediately follows from the fact that the depth of a binary tree with n leaves cannot be smaller than $\lceil \log_2 n \rceil$. ■

B. Arrival Times

Arrival times at the primary inputs of a circuit can be taken into account by redefining the depth of a BDAG as follows:

$$\mathcal{D}(T) = \begin{cases} \mathrm{AT}(T), & \text{if } T.\mathrm{op} = \perp \\ 1 + \max\left(\mathcal{D}(T.\mathrm{left}), \mathcal{D}(T.\mathrm{right})\right), & \text{otherwise} \end{cases}$$

where $\mathrm{AT}(T)$ is the arrival time of the primary input associated to the leaf node T.

The lower bound on the depth of a BDAG can now be recalculated taking into account arrival times at the primary inputs. For that, one can assume that an input with arrival time $AT(T)$ can be represented as a tree with $2^{AT(T)}$ leaves. This tree mimics the delay of the input.

Theorem 3 (Lower bound on depth): Let G' be a BDAG obtained from G by applying the ACD-rules. A lower bound for $\mathcal{D}(G')$ is

$$\mathcal{D}(G') \geq \left\lceil \log_2 \sum_{p \in \mathrm{paths}(G)} 2^{\mathrm{AT}(p)} \right\rceil \tag{3}$$

where $AT(p)$ is the arrival time of the primary input associated to the leaf of path p in G.

Proof: The proof is similar to that of Theorem 2. We first prove the result for the particular case in which all arrival times are zero except for one, which is a (see Fig. 5). We also assume that a is a natural number. It is easy to see that a perfectly balanced tree with depth n will have at most $2^n - 2^a + 1$ leaves, which comes from the sum

$$2^{n-1} + 2^{n-2} + \cdots + 2^{a+1} + 2^a + 1$$

(the term 1 corresponds to the leaf with late arrival, as shown in Fig. 5). The expression above can be rewritten as

$$2^a(2^{n-a-1} + 2^{n-a-2} + \cdots + 2 + 1) + 1 =$$
$$2^a(2^{n-a} - 1) + 1 = 2^n - 2^a + 1$$

which indicates that a leaf with arrival time a accounts for 2^a leaves with arrival time zero. This result can be extended to multiple leaves with nonzero arrival times, thus leading to the inequality (3). The extension to real numbers for the arrival times is straightforward. ■

Note that the expression (3) reduces to expression (2) when $AT(p) = 0$, for any p.

C. BDAG Representation

The algorithms presented in this paper have been implemented in a data structure to represent circuits with two-input gates. This representation is very similar to that of Boolean Expression Diagrams [18]. A common manager represents all BDAGs and a single instance of each sub-BDAG in the manager is guaranteed. Internal nodes only represent AND operators and edges can be complemented to represent negations. For the sake of simplicity, in this paper we will still distinguish between AND and OR nodes when depicting circuits.[1]

The BDAG manager also has cache tables to speed up the recursive algorithms that traverse the circuits from top to bottom in such a way that operations on reconvergent paths are not repeated. The details of the cache management are not shown in the algorithms presented in this paper.

The circuits in the manager are also organized in *equivalence classes*. An equivalence class is a list of circuits that are known to be equivalent. For example, assume that X, Y, and Z are circuits in the manager and that there also exists another circuit $C_1 = X(Y + Z)$. This circuit belongs to the equivalence class $[C_1]$. After applying the distributive law, the following circuit can be obtained $C_2 = XY + XZ$. Assume that C_2 already existed in the manager with its own equivalence class $[C_2]$. Since C_1 and C_2 are now known to be equivalent, both classes are merged in such a way that $[C_1] = [C_2] = [C_1] \cup [C_2]$.

In practice, equivalence classes are implemented as chained lists that occupy one pointer in each node. For efficiency reasons, the list is ordered by depth and size of the BDAGs. An auxiliary table keeps track of all equivalence classes in the manager in such a way that knowing whether two functions are in the same class takes constant time. Equivalent functions under complementation are also kept in the same class.

IV. Algorithms for Time Optimization Using ACD Rules

This section presents algorithms for the exploration of BDAGs aiming at reducing their depth. First, algorithms for minimal depth by using only the AC-rules are presented. Next, an algorithm incorporating the D-rule is proposed.

A. Minimal-Delay Clusters (AC-Rules)

The topmost cluster of G is the set of sub-BDAGs closer to the root that have an operation different from G. Formally, the topmost cluster of a BDAG is obtained by the algorithm CLUSTER in Fig. 6.

Given a cluster, a minimum-delay tree can be built by combining the elements of the cluster in an appropriate way, trying the tallest subtrees to be closer to the root. Baer and Boven [19] proposed an algorithm to build such a tree. It is an iterative algorithm that maintains all elements of the cluster in a priority

[1]This distinction is also maintained in the package by properly keeping track of the complemented edges found in the paths.

```
CLUSTER (G)
   { Pre-cond: G.op ≠ ⊥. Returns a set of sub-BDAGs }
   If G.op = G.left.op then C_L := CLUSTER(G.left);
   else C_L := G.left;
   If G.op = G.right.op then C_R := CLUSTER(G.right);
   else C_R := G.right;
   return C_L ∪ C_R;
```

```
MIN_DELAY_CLUSTERS (G)
   { Returns a BDAG with min-delay clusters }
   { Q is a list ordered by depth }
   if G.op = ⊥ then return G;
   C := CLUSTER(G); Q := ∅;
   for each c ∈ C do
      INSERT (Q, MIN_DELAY_CLUSTERS(c));
   while |Q| > 1 do
      X := EXTRACT_MIN_DEPTH(Q);
      Y := EXTRACT_MIN_DEPTH(Q);
      INSERT(Q, (G.op X Y));
   return EXTRACT(Q);
```

Fig. 6. Algorithm for minimum-delay clusters.

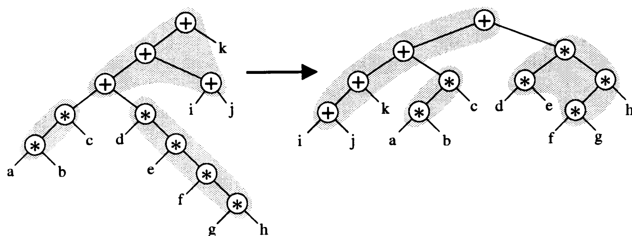

Fig. 7. Application of MIN_DELAY_CLUSTERS.

queue ordered by the depth of the elements. At each iteration, the two shortest elements are extracted and a new tree is built and inserted in the queue. The algorithm terminates when only one element is left in the queue, which is the returned tree. This simple algorithm was proven to be optimal in [20]. It is also the algorithm used in SIS for minimum-delay decomposition of AND and OR gates [6], though no proof of optimality was given.

The algorithm MIN_DELAY_CLUSTERS to obtain a minimum-delay BDAG by only using the associative and commutative laws is shown in Fig. 6. The algorithm was proposed in [20] and was proven to minimize delay. It is a recursive algorithm that invokes the algorithm by Baer and Boven to build minimum delay clusters (the "while" loop).

Fig. 7 depicts an example on the solution derived by the algorithm. The shadowed areas correspond to the clusters visited when traversing the tree. Note that the algorithm produces another tree with the same size, since the associative and commutative laws do not change the size of the tree. This also implies that $\Pi(G)$ remains the same, although the size of G may vary (increase or decrease) if the sharing of reconvergent paths is modified.

B. Distributive Law (D-rule)

The distributive law can only be applied to two nodes of a BDAG, n_1 and n_2, for which the following condition holds:

$$n_2 \in \text{CHILDREN}(n_1) \quad \wedge \quad n_1.\text{op} \neq n_2.\text{op} \quad \wedge \quad n_2.\text{op} \neq \bot \,.$$

The transformation is shown in Fig. 8. By itself, the distributive law cannot provide any performance improvement, since

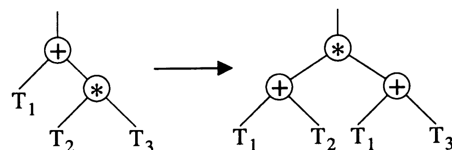

Fig. 8. Distributive law.

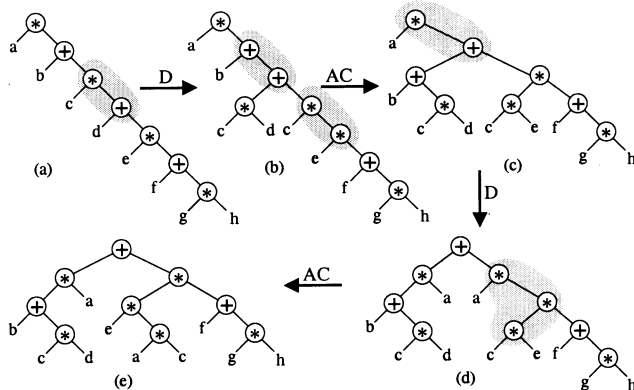

Fig. 9. Application of ACD rules to optimize performance.

the depth of the resulting BDAG is not shorter than the depth of the original BDAG. It can even produce some performance degradation if

$$\mathcal{D}(T_1) > \max\left(\mathcal{D}(T_2), \mathcal{D}(T_3)\right).$$

However, the distributive law changes the structure of the clusters and enables the application of AC-rules that can potentially result in shorter depths. The combination of D- and AC-rules is illustrated in the example of Fig. 9. After the application of a D-rule, a minimum-delay tree is obtained by running the MIN_DELAY_CLUSTERS algorithm (AC-rules).

C. ACD_Speed

The solution in Fig. 9(e) can only be obtained by applying the D-rule to certain nodes of the tree. One can immediately see that this solution cannot be obtained if the D-rule is applied to the root node of Fig. 9(a). Therefore, the order in which rules are applied is relevant for searching optimal solutions.

Fig. 10 presents an algorithm for speeding-up a BDAG by using ACD-rules. It assumes that F is an initial BDAG with minimal number of nodes, e.g., obtained by area minimization transformations on a Boolean network. The required time, in terms of number of logic levels, is also another parameter. The algorithm implements a dynamic programming approach with memoization that alternatively applies the D-rule to one of the nodes and MIN_DELAY_CLUSTERS to the BDAG. The set `explored` collects all the solutions generated in the algorithm.

In order to control the explosion of solutions, a frontier with limited width is selected at each layer of the search. The width of the frontier k is a factor that can be tuned according to the exhaustiveness of the search. The selection of "best" solutions is done by giving priority first to delay (depth of the BDAG) and second to area (size of the BDAG).

Lower_Bound_Depth(F) calculates a lower bound on the depth of the circuit. It corresponds to expression (3). The

```
ACD_SPEED (F, ReqTime)
  { F is a BDAG. Returns a BDAG }
  { ReqTime is the required time (in logic levels) }
  Best := MIN_DELAY_CLUSTERS (F);
  MaxTime := MAX(ReqTime, Lower_Bound_Depth(F));
  frontier := {Best}; explored := {Best};
  while depth(Best) > MaxTime  ∧  "improving" do
    new := ∅;
    for each F_r ∈ frontier do
      for each node n ∈ F_r
      such that D-rule is applicable do
        F' := APPLY_DISTRIBUTIVE (F_r, n);
        F'' := MIN_DELAY_CLUSTERS (F');
        if F'' ∉ explored then
          explored := explored ∪ {F''};
          new := new ∪ {F''};
          Best := Best_Delay_Area (Best, F'');
    frontier := Select_Best_k_Circuits (new, k);
  return Best;
```

Fig. 10. Algorithm for speeding-up.

algorithm stops when the depth of the best circuit is not larger than the maximum of the required time and the lower bound on depth. The calculation of this bound contributes to prune the exploration significantly. The algorithm also stops when no improvement has been observed during a few iterations. The "improvement" criterion is another tunning parameter of the algorithm.

V. Logic Decomposition

The decomposition of a Boolean function F is performed recursively from root to leaves by finding an operation op and two functions, F_1 and F_2, such that $F = F_1 \text{ op } F_2$. This type of decomposition was originally called quasi-algebraic decomposition [21] and often referred to as bi-decomposition [11], [12], [22]. Each level of recursion defines a logic level of the function.

The main algorithm is shown in Fig. 11. In order to improve the quality of the search, different bi-decomposition methods can be used in the same framework. The actual implementation uses two methods, hidden in the function `Decompose_2input_gates`. One of them is based on finding algebraic factored forms and the other is based on finding BDD approximations.

The recursive paradigm behind the ACD_DECOMPOSE algorithm interleaves the generation of bi-decompositions with the speed optimization by means of ACD_SPEED. The function `Decompose_2input_gates` works in two steps.

1) It finds a bi-decomposition F_1 op F_2 of the incompletely specified function defined by (ON,DC), where F_1 and F_2 are now completely specified functions. The bi-decomposition is performed by one of the methods explained in Section V-A.
2) It decomposes F_1 and F_2 into a factored form of two-input operators by using fast methods for algebraic factorization. This step is an attempt to find a reasonable representation of the functions and estimate their delay.

After these two steps, the two-input netlist is optimized for speed (F_s := ACD_Speed(F_d, ReqTime)). The parameter

```
ACD_DECOMPOSE (ON, DC, ReqTime)
  { ON and DC are covers. Returns a tree }
  G_1 := BI-DECOMP (ON, DC, ReqTime, method_1);
  :
  G_n := BI-DECOMP (ON, DC, ReqTime, method_n);
  G := Choose_Best_BDAG (G_1, ..., G_n);
  F_l := collapse (G.left); {cover of the left subgraph}
  F_r := collapse (G.right); {cover of the right subgraph}
  if D(G.right) > D(G.left) then swap(F_l, F_r);

  {Decompose the fastest child (left)}
  D_l := ACD_DECOMPOSE (F_l, DC, ReqTime -1);

  {Update DC for the slowest child of the tree}
  F_l := collapse (D_l); {cover of the left subtree}
  if G.op = AND then
    F_r := F_r · F_l;  DC = DC + ¬F_l;
  else {G.op = OR}
    F_r := F_r · ¬F_l; DC = DC + F_l;

  {Decompose the slowest child of the tree}
  D_r := ACD_DECOMPOSE (F_r, DC, ReqTime -1);
  return (G.op, D_l, D_r);
```

```
BI-DECOMP (ON, DC, ReqTime, method)
  { ON and DC are covers. Returns a tree }
  { "method" determines the decomposition strategy }
  F_d := Decompose_2input_gates (ON, DC, method);
  F_s := ACD_Speed (F_d, ReqTime);
  return F_s;
```

Fig. 11. Algorithm for logic decomposition.

`ReqTime` defines the desired required time for the function. `ReqTime` is measured in logic levels and it is decreased each time a new recursion level is invoked.

The main algorithm, ACD_DECOMPOSE, chooses the best BDAG G obtained from all bi-decompositions. This selection is done by first giving priority to speed. If the required time is met by several BDAGs, the one with the smallest area (the number of nodes) is selected. At this point, the root node of G determines the operator for a new level of logic. The rest of the netlist is collapsed and prepared for a new level of recursion. After the decomposition has been done for one of the children (D_l), the observability dc is calculated for the other. As an example, in case the topmost operation is an AND, the dc-set for F_r is enlarged when F_l is zero. Since the definition of the dc-set for each children depends on the order in which they are decomposed, the slowest one is always decomposed last. In this way, it has more chances to have a larger dc-set.

The satisfiability and observability dcs calculated at each node of the tree are propagated down during the recursive decomposition.

A. Bi-Decomposition Methods

Two bi-decomposition methods are used in the actual implementation of the decomposition algorithm.

The first is a factorization based on the search of kernels and algebraic division [17]. In the current implementation, this factorization is implemented by the function `factor_good` in SIS [23].

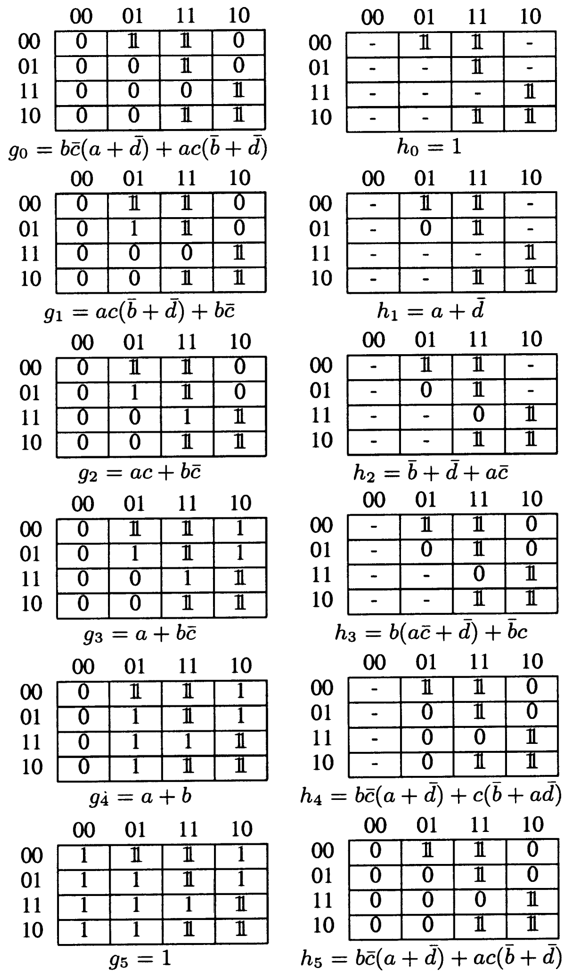

Fig. 12. Conjunctive decomposition by function approximations.

The second approach uses the power of Boolean algebra and is computationally more expensive. It is based on the calculation of function approximations. Fig. 12 depicts an example of the approach for the conjunctive decomposition[2] of a function f. The cells with label 𝟙 denote the original ON-set of the function. The cells with label 1 represent over-approximations. The aim of the method is to calculate two functions g and h such that $f = g \cdot h$. A necessary condition is that

$$f \subseteq g \quad \wedge \quad f \subseteq h.$$

The method iteratively calculates over-approximations g_i of f and the associated conjuncts h_i by Boolean minimization using the observability dc derived from g_i. The more accurate the approximation g_i is, the larger the dc is to minimize h_i. The K-maps in Fig. 12 represent a sequence of approximations starting with an initial exact approximation g_0.

The actual method presented in this paper uses BDDs to calculate function approximations; it is inspired on the approach presented in [24]. Fig. 13 presents an example for the same function depicted in Fig. 12. The approach consists of remapping some nodes of f in such a way that the BDD size is reduced but the number of minterms of the new BDD is not increased too

[2]The approach for disjunctive decomposition is similar, but using under-approximations instead.

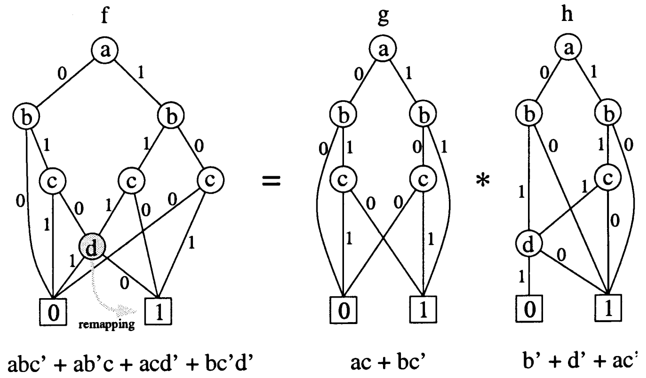

Fig. 13. BDD-based decomposition.

much (a *dense* over-approximation). In the figure, the approximation is calculated by remapping the node d into the constant 1. f is reduced by two nodes (g) and the number of minterms is increased by two. Once g is known, h can be calculated by BDD minimization: $g \subseteq h \subseteq f + \bar{g}$. This process is iteratively executed to generate a sequence of approximations as in Fig. 12. A cost function based on BDD sizes is used to select one of the approximations (g_i, h_i) in the sequence.

The actual BDD-based approach used in this paper is similar to the one in [24], but considers many more nodes as candidates for replacement (same level, children, and grandchildren).

It is important to notice that the approximation approach subsumes the conjunctive and disjunctive bi-decompositions proposed by other authors [22], [25] in which the BDD transformations can be reduced to remapping some nodes into constants or other nodes of the same BDD. Only the particular heuristics used in each approach may lead to different decomposition results in practice.

Iterative Calculation of Observability Don't Cares (ODC): By observing the example in Fig. 12, it is easy to realize that the ODC for g_i can be recalculated after the minimization of h_i. This process can be repeated until a satisfactory solution is found. The following loop could be executed to improve a conjunctive decomposition $f = g \cdot h$ with a given satisfiability dc:

```
repeat
  dc = SDC ∪ h;
  g = miminize(g, dc);
  dc = SDC ∪ g;
  h = miminize(h, dc);
until no improvement.
```

In practice, the experimental results have shown that the initial decomposition is rarely improved by this loop.

VI. Experimental Results

The strategy presented in this paper has been implemented in SIS. The results have been compared with SIS and the method for bi-decomposition presented in [11]. The experiments have been run on a subset of small and medium size Microelectronics Center of North Carolina benchmarks. Table I describes the

TABLE I
SCRIPTS USED FOR THE EXPERIMENTAL RESULTS

			ACD		
algebraic	**rugged**	**bidec** [11]	**tree**	**DAG**	**cluster**
collapse					
algebraic*4	rugged*4	bidecomp	acd_decompose		
speed_up				resub -a	
					cl_collapse; fx -z
map -n1 -AFG (library lib2.genlib)					

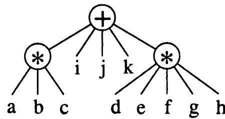

Fig. 14. Cluster collapsing (cl_collapse) of the circuit in Fig. 7.

scripts used for the experiments. The suffix "*4" indicates that the script has been run four times.[3]

All the benchmarks were multilevel netlists. Initially, the circuits were collapsed and converted into two-level forms. After that, the algebraic script algebraic was the one deriving the best results for SIS. The scripts ACD-* are the ones implementing the strategy of this paper. ACD − tree derives a tree decomposition (no sharing between isomorphic subtrees). ACD − DAG transforms the tree into a DAG by sharing all isomorphic subtrees. This is achieved by algebraic resubstitution.

Finally, ACD − cluster also tries to share common subexpressions within the final clusters of the DAG. For example, if one cluster implements the expression $(a+b)+(c+d)$ and another implements $(a+(e+(c+f))$, they will be re-expressed as $(a+c)+b+d$ and $(a+c)+e+f$, sharing $a+c$, even though the depth of the circuit can be increased by sharing the common subexpressions. This is achieved by collapsing all clusters in the DAG (command cl_collapse, see Fig. 14) and extracting common cube divisors.

Table II reports the results. After logic decomposition, all the circuits have been mapped into the library lib2, which includes a rich set of static CMOS gates up to four-input NAND/NOR and six-input AOI/OAI gates. The column L reports the number of levels of the circuit before technology mapping, counted as the depth of the circuit represented with two-input gates (inverters are ignored).

ACD − tree obtains a 20% delay reduction at the expense of 35% area increase. If sharing is allowed (ACD − DAG and ACD − cluster) the delay reduction is more moderate (11% and 8%, respectively), but area is significantly better (only 8% and 4% increase, respectively). The delay increase of ACD − DAG and ACD − cluster with regard to ACD − tree is due to two factors, mainly: 1) the capacitive load of the shared nodes and 2) suboptimality of the tree-mapping algorithm when working on DAGs. In some circuits, (e.g., 9symml and frg1) area is drastically reduced due to the power of Boolean bi-decomposition.

It is important to emphasize that the ACD scripts could reduce, on average, almost one level of logic with regard to algebraic $(6.36 \rightarrow 5.49)$.

[3]Experimentally, we found this number to be adequate to obtain good-quality results.

Fig. 15 plots the normalized average results. It is interesting to see that rugged and bidec produce suboptimal results for the area/delay tradeoff. It is also important to observe that there is a potential space of configurations between the points ACD − DAG and ACD − tree. This space can be explored by partially sharing the isomorphic subtrees produced by ACD-tree (e.g., by sharing subtrees in noncritical paths only). We believe that results with delay similar to ACD − tree and area similar to ACD − cluster could be obtained by using DAG covering [26] and gate duplication techniques [27] during technology mapping.

The overall CPU time of the ACD algorithms is about a factor of two the CPU time of the algebraic script, and comparable to that of the rugged script. In algebraic, rugged and bidec, most of the CPU time is spent on the speed_up command, whereas the ACD scripts evenly distribute the effort between finding decompositions and balancing them.

The results were all obtained by collapsing the whole network. This brute-force approach cannot be applied for large networks. In the future, we foresee combining partial collapsing and decomposition to manage much larger examples. The two largest examples that we decomposed were apex6 (135 inputs, 99 outputs, and 803 gates) and vda (17 inputs, 39 outputs, and 1237 gates).

Results were also obtained with the algebraic script without collapsing the netlist, i.e., transforming the original netlist described by the benchmark. The results were, in general, worse than those obtained by collapsing.

A. Impact of Different Bi-decomposition Methods

The current approach for bi-decomposition combines two methods, algebraic and BDD-based approximations, but which is the impact of each method on the quality of the bi-decompositions? To study this impact, the script ACD − tree was run with different bi-decomposition methods. The results are summarized in Table III.

The first two rows report the results obtained by using the algebraic bi-decomposition without and with the BDD-based bi-decomposition, respectively. The contribution of the BDD-based bi-decomposition is manifested in the reduction of number of levels and delay after technology mapping.

It is important to mention that the BDD-based decomposition is only chosen in a small percentage of times (typically between 5% and 10%, depending on the example). In many cases, the BDD-based decomposition derives the same solution as the algebraic decomposition. But the most interesting aspect is that the contribution of the BDD-based decomposition is more important at the topmost nodes of the tree, when the function is still complex and offers many different possibilities for nonalgebraic decompositions. It is at the topmost levels when the decisions have a more tangible impact on the final solution. The decompositions close to the leaves of the tree are almost always algebraic, especially when the functions become unate.

Another experiment was also performed to check the contribution of the method proposed in this paper with regard to the approximation method presented in [24], subsumed by the former. The results reported in the third row of Table III show that, although small, there is still a contribution of the new method in number of levels and delay.

TABLE II
EXPERIMENTAL RESULTS

circuit	PI	PO	algebraic			rugged			bidec			ACD-tree			ACD-DAG		ACD-cluster	
			L	D	A	L	D	A	L	D	A	L	D	A	D	A	D	A
9symml	9	1	11	10.43	195.0	11	1.02	0.75	11	1.06	0.66	8	0.67	0.46	0.71	0.35	0.70	0.32
alu2	10	6	11	9.99	335.7	14	1.41	0.71	12	1.19	1.20	10	0.83	1.11	0.86	0.97	0.88	0.93
apex6	135	99	11	11.61	649.3	15	1.37	1.02	9	0.78	1.26	7	0.63	1.63	0.72	1.26	0.74	1.20
apex7	49	37	9	9.90	257.7	12	1.28	1.05	12	1.23	2.37	8	0.82	2.21	0.89	1.15	0.90	1.06
b1	3	4	3	2.86	9.0	3	1.00	1.00	3	1.18	1.07	3	1.00	0.93	0.99	0.85	0.95	0.96
b9	41	21	6	6.80	109.7	7	1.10	0.95	7	1.11	1.26	6	0.76	1.48	0.91	0.99	0.91	1.02
c8	28	18	7	6.99	108.3	6	1.00	1.00	6	0.84	1.20	6	0.78	1.40	0.97	1.07	1.06	1.07
cc	21	20	5	5.91	56.7	5	1.11	0.90	5	1.09	1.22	4	0.89	1.40	0.94	1.14	1.06	1.11
cht	47	36	4	6.03	121.7	4	1.02	1.08	4	0.86	1.79	4	0.76	1.56	0.86	1.44	1.00	1.48
cm138a	6	8	3	4.68	22.3	3	1.03	1.16	3	0.97	1.09	3	0.79	1.66	0.82	1.69	0.99	1.22
cm150a	21	1	8	7.25	45.7	7	0.84	1.18	7	1.02	1.52	7	0.82	1.20	0.88	1.18	0.89	1.01
cm151a	12	2	6	5.80	21.3	6	1.01	1.20	6	1.19	1.84	6	0.97	2.20	0.99	1.03	1.00	1.19
cm152a	11	1	5	4.96	23.0	5	1.02	1.07	6	1.09	1.39	5	1.02	0.99	1.01	1.01	1.03	1.00
cm162a	14	5	5	5.93	60.0	6	1.06	0.71	6	1.13	0.99	5	0.76	0.96	0.82	0.78	0.93	0.83
cm163a	16	5	6	6.19	44.0	5	0.92	0.98	6	0.96	1.06	5	0.69	1.10	0.89	1.11	0.92	1.02
cm42a	4	10	2	4.21	27.3	2	1.00	1.00	2	0.95	1.11	2	0.75	0.95	0.89	1.00	0.95	0.95
cm82a	5	3	5	5.24	24.3	7	1.29	1.23	5	0.96	0.92	5	0.85	1.15	0.99	0.77	0.93	0.79
cm85a	11	3	6	5.57	49.0	6	1.07	0.95	7	1.17	0.99	5	0.80	0.88	0.86	0.82	0.86	0.86
cmb	16	4	4	4.85	27.7	4	1.00	1.00	4	0.91	1.07	4	0.86	1.43	1.00	1.18	1.01	1.13
count	35	16	7	8.65	186.7	7	1.00	0.93	7	0.85	1.17	7	0.80	1.76	0.97	0.96	1.02	0.88
cu	14	11	7	5.37	46.0	6	1.20	1.40	6	1.04	1.28	5	0.84	1.51	1.00	1.30	0.99	1.22
f51m	8	8	8	7.80	122.3	15	2.03	0.97	9	1.12	0.63	7	0.81	1.20	0.87	1.02	0.85	1.01
frg1	28	3	11	9.17	100.7	10	0.95	1.26	7	0.70	0.57	7	0.58	0.57	0.63	0.46	0.62	0.55
i1	25	16	6	5.80	40.7	5	0.96	1.03	6	0.91	1.33	5	0.70	1.36	0.79	1.20	0.87	1.30
lal	26	19	6	7.25	90.7	10	1.55	1.46	6	0.90	1.21	6	0.74	1.67	0.87	1.04	0.89	1.03
majority	5	1	4	3.06	6.0	4	1.00	1.00	4	0.92	0.89	4	0.89	0.89	0.89	0.89	0.89	0.89
mux	21	1	8	7.71	44.7	7	0.82	1.15	10	1.19	1.95	6	0.79	1.36	0.78	1.33	0.79	1.16
pcle	19	9	6	6.37	75.3	6	1.08	0.98	7	1.07	1.42	6	0.87	1.31	0.97	1.09	1.04	1.00
pcler8	27	17	6	7.70	114.7	7	1.04	0.72	7	0.96	1.40	5	0.73	1.57	0.78	1.19	0.85	1.06
pm1	16	13	5	5.25	38.3	5	1.21	1.43	4	0.88	1.49	4	0.79	1.56	0.83	1.33	1.01	1.10
sct	19	15	6	6.76	66.0	9	1.49	1.76	6	0.95	1.57	5	0.72	1.80	1.05	1.18	0.99	1.17
tcon	17	16	2	3.65	27.0	2	1.00	1.00	2	1.00	1.00	2	0.92	1.02	0.97	1.07	0.99	1.07
term1	34	10	10	9.50	107.0	8	0.87	1.16	11	1.12	2.06	8	0.74	2.27	0.76	1.64	0.83	1.49
ttt2	24	21	7	8.66	166.3	17	2.21	1.37	7	0.92	1.34	6	0.75	1.46	0.89	1.04	0.97	0.98
unreg	36	16	4	6.07	111.3	4	0.97	0.98	4	1.04	1.02	4	0.77	1.01	0.77	0.95	0.77	1.00
vda	17	39	9	11.42	743.7	14	1.42	0.98	13	1.16	1.94	8	0.79	2.02	0.83	1.37	0.87	1.30
x1	51	35	7	6.78	263.3	7	1.05	1.00	9	1.21	1.45	6	0.86	1.33	0.89	1.04	0.97	0.95
x2	10	7	6	5.20	45.7	7	1.38	1.20	7	1.44	1.09	5	0.86	1.01	0.87	1.10	0.92	1.02
z4ml	7	4	6	6.27	44.0	8	1.21	0.83	7	0.99	1.02	5	0.79	1.42	0.94	0.95	0.89	0.98
average			**6.36**	**1.00**	**1.00**	**7.33**	**1.15**	**1.07**	**6.61**	**1.03**	**1.27**	**5.49**	**0.80**	**1.35**	**0.89**	**1.08**	**0.91**	**1.04**

PI: primary inputs; PO: primary outputs; L: depth (levels of two-input gates); D,A: delay and area (normalized with respect to algebraic). The area in algebraic has been divided by the area of a NAND2 gate. The average values for delay and area in algebraic have also been normalized to 1.00. The number of levels for the three ACD methods is the same.

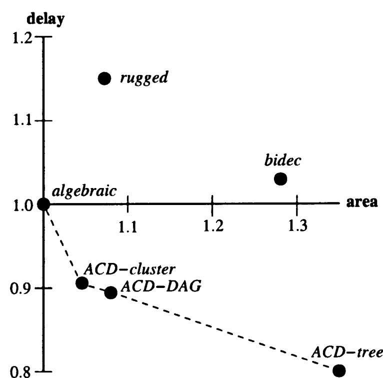

Fig. 15. Summary of results in Table I.

TABLE III
IMPACT OF BI-DECOMPOSITION METHODS USING THE ACD-TREE SCRIPT

Bi-decomposition methods	Average results		
	L	D	A
only algebraic	5.64	0.82	1.36
algebraic + approx. (this paper)	5.49	0.80	1.35
algebraic + approx. [24]	5.64	0.81	1.36

VII. RELATED WORK AND DISCUSSION

Speed_up [6] uses a strategy of partial collapsing and resynthesis of critical paths. Resynthesis of each node is performed by extracting kernel-based divisors that reduce the arrival time at the output.

Restructuring by applying the ACD rules works at a finer level of granularity than kernel-based decomposition and may potentially lead to better results, as it was illustrated by the examples in Section II-A. Additionally, the ACD_SPEED algorithm has been designed in such a way that it can explore more solutions even though no global improvement has been observed during few iterations. These two features result in a better exploration of the space of solutions, at the expense of more computational cost. However, the experimental results show that this cost is still affordable.

A. Sharing Before Reducing Depth

The experimental results also manifest the problems of speeding up networks that have been highly optimized for area. The results obtained by the rugged script are inferior,

on average, than those obtained by the `algebraic` script. As an example, we took `apex6` from the benchmark suite and compared the networks before executing the `speed_up` command. Here are the results:

	algebraic		rugged	
	nodes	levels	nodes	levels
before speed_up	718	14	711	22
after speed_up	733	11	750	15

The algebraic script initially derives a slightly larger netlist (718 nodes, each node is a two-input gate) with regard to the rugged script (711 nodes). However, the number of logic levels is much higher for the `rugged script`, due to the more aggressive sharing. This fact has a tangible impact when trying to speed up the netlist. The result obtained by the rugged script ends up having a larger number of nodes and logic levels. This example, in particular, and the average results in Table II illustrate the phenomenon mentioned in the introduction of this paper (Fig. 1).

B. Logic Decomposition During Technology Mapping

In [9], a combined approach for logic decomposition and technology mapping was proposed. The strategy consists of generating all possible decompositions of a circuit and representing them compactly encoded in a graph. The decompositions are generated by applying local transformations that correspond to the ACD rules described in this paper.[4]

In principle, one might think that the exploration power of both techniques is the same, except for the inaccuracy introduced by heuristics. However, the approach in [9] only uses the D-rule in its factoring direction, i.e.,

$$ab + ac \quad \longrightarrow \quad a(b + c).$$

This limitation is crucial for the reduction of the depth of a circuit. As an example, the circuit in Fig. 9(a) would never be changed by the local transformations in [9], i.e., the closure of the circuit would be itself. On the other hand, the application of the D-rule in its expanding form enables the exploration of more efficient solutions, as shown in Fig. 9(b)–(e).

The incorporation of the expanding form of the D-rule, increases the exploration space exponentially. Given a known upper bound on the minimum depth of the circuit,[5] all trees up to 2^d nodes could potentially lead a solution with depth not larger than d, as shown in Theorem 2. This is the main reason why an exhaustive exploration can be computationally expensive and the heuristic search of the ACD_SPEED algorithm (Fig. 10) is proposed.

To emphasize the difference between both approaches, we run few small benchmarks with an implementation of the technology mapping algorithm in [9].[6] As an example, the results for the `algebraic` and `ACD − cluster` scripts for `cm82a` with this technology mapper are the following:

algebraic with graph map [9]		ACD-cluster with graph map [9]	
delay	area	delay	area
4.63	23.3	4.35	18.0

This result confirms that: 1) the algorithm by Lehman and Watanabe can find better solutions than a conventional tree mapper (see the corresponding result in Table I) and 2) the technique presented in this paper is not subsumed by Lehman and Watanabe's approach.

VIII. Conclusion

This paper has presented an approach for decomposing logic functions. It aims at reducing the number of logic levels of the network and succeeds in doing so for many examples, compared with previous existing techniques. However, there are still many questions on the air: how far are we from optimum solutions? Would it be possible to calculate tight lower/upper bounds on the depth of a circuit implementing a Boolean function? How much area must we pay to reduce one logic level? More research is needed in this direction.

This paper has shown that the area/delay tradeoff can be further explored and tangible improvements can still be obtained with regard to previous techniques.

Acknowledgment

The author wishes to thank the reviewers for their suggestions to improve the paper.

References

[1] M. Fujita and R. Murgai, "Delay estimation and optimization of logic circuits: a survey," in *Proc. Asia South Pacific Design Automation Conf.*, 1997, pp. 25–30.

[2] R. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, Univ. Calif., Berkeley, Apr. 1989.

[3] J. Vasudevamurthy and J. Rajski, "A method for concurrent decomposition and factorization of Boolean expressions," in *Proc. Int. Conf. Computer-Aided Design*, 1990, pp. 510–513.

[4] K. Barlett, R. Brayton, G. Hachtel, R. Jocoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 723–740, June 1988.

[5] K. Chen and S. Muroga, "Timing optimization for multi-level combinational circuits," in *Proc. ACM/IEEE Design Automation Conf.*, 1990, pp. 339–344.

[6] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1988, pp. 282–285.

[7] H. Touati, H. Savoj, and R. Brayton, "Delay optimization of combinational circuits by clustering and partial collapsing," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 188–191.

[8] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1988.

[9] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 813–834, Aug. 1997.

[10] R. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory Switching*, vol. 29, 1959, pp. 74–116.

[11] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proc. ACM/IEEE Design Automation Conf.*, June 2001, pp. 282–285.

[4]The inverter transformations in [9] are naturally covered by the complemented arcs in the BDAGs.

[5]An upper bound can always be found by taking the depth of a trivial implementation, e.g., a sum-of-products implemented with two-input gates.

[6]A prototype was designed by [28]. Only very small examples were executed due to the complexity of the algorithm and the naive implementation of some data structures in the first prototype of the algorithm.

[12] S. Yamashita, H. Sawada, and A. Nagoya, "New methods to find optimal nondisjoint bi-decompositions," in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 59–68.
[13] S.-C. Chang, M. Marek-Sadowska, and T. Hwang, "Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1226–1235, Oct. 1996.
[14] T. Sasao, "FPGA design by generalized functional decomposition," in *Logic Synthesis and Optimization*. Norwell, MA: Kluwer, 1993, pp. 233–258.
[15] C. Scholl, *Functional Decomposition With Application to FPGA Synthesis*. Norwell, MA: Kluwer, 2001.
[16] D. Kuck, *The Structure of Computers and Computation*. New York: Wiley, 1978.
[17] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. Int. Symp. Circuits Syst.*, May 1982, pp. 49–54.
[18] H. Andersen and H. Hulgaard. Boolean expression diagrams. presented at *IEEE Symp. Logic Comput. Sci.*. [Online] citeseer.nj.nec.com/andersen97boolean.html
[19] J. Baer and D. Bovet, "Compilation of arithmetic expressions for parallel computations," in *Proc. IFIP Congress* North-Holland, The Netherlands, 1968, pp. 340–346.
[20] J. Beatty, "An axiomatic approach to code optimization for expressions," *J. Assoc. Comput. Mach.*, vol. 19, no. 4, pp. 613–640, Oct. 1972.
[21] T. Stanion and C. Sechen, "Quasialgebraic decompositions of switching functions," in *Proc. 16th Conf. Adv. Res. VLSI*, 1995, pp. 358–367.
[22] C. Yang, M. Ciesielski, and V. Singhal, "BDS: a BDD-based logic optimization system," in *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 92–97.
[23] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS : A System for Sequential Circuit Synthesis," Tech. Rep. Univ. Calif., Berkeley, May 1992.
[24] K. Ravi, K. McMillan, T. Shiple, and F. Somenzi, "Approximation and decomposition of binary decision diagrams," in *Proc. Design Automation Conf.*, 1998, pp. 445–450.
[25] Y.-T. Lai, K.-R. Pan, and M. Pedram, "OBDD -based function decomposition: algorithms and implementation," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 977–990, Aug. 1996.
[26] Y. Kukimoto, R. Brayton, and P. Sawkar, "Delay-optimal technology mapping by DAG covering," in *Proc. Design Automation Conf.*, 1998, pp. 348–351.
[27] A. Srivastava, R. Kastner, and M. Sarrafzadeh, "Timing driven gate duplication: complexity issues and algorithms," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 447–450.
[28] D. Bañeres, Algorithm for logic decomposition and technology mapping, Facultat d'Informàtica de Barcelona, Barcelona, Spain, July 2002.

Jordi Cortadella (S'87–M'88) received the M.S. and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 1985 and 1987, respectively.

He is a Professor in the Department of Software, Universitat Politècnica de Catalunya. In 1988, he was a Visiting Scholar at the University of California, Berkeley. His research interests include formal methods and computer-aided design of very large scale integration systems with special emphasis on asynchronous circuits, concurrent systems, and logic synthesis. He has coauthored over 100 research papers in technical journals and conferences.

Dr. Cortadella has served on the technical committees of several international conferences in the field of Design Automation and Concurrent Systems. He served as a Symposium Co-Chair for the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems.