Support-Reducing Functional Decomposition for FPGA Technology Mapping

Lucas Machado and Jordi Cortadella
Department of Computer Science, Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract—**The goals of decomposition and optimization methods in technology-independent logic synthesis have historically been the reduction of cubes and literals, or nodes and levels of a directed-acyclic graph. These cost functions have a high correlation with mapped circuits composed of standard cells, in which the simplest logic block has 2 inputs. However, this correlation does not hold for look-up table based field-programmable gate arrays (FPGAs), which are composed of logic blocks with *k* inputs (typically 4 to 6). Also, local optimizations have limited power due to the structural bias of circuit descriptions, as these are generally targeted to standard cells.**

This paper tries to reduce the structural biasing by iteratively collapsing the subject network and decomposing the derived functions using the support as cost function. The proposed method improves the FPGA mapping results of a commercial tool for the 20 largest MCNC benchmarks, with gains of 27% in delay plus 10% in area when targeting delay; and a reduction of 16% in area plus 12% in delay with area as cost function. Moreover, 12 of the best known results for delay (and 3 for area) of the EPFL benchmarks are updated.

I. Introduction

Field-programmable gate arrays (FPGAs) are integrated circuits consisting of programmable logic blocks and reconfigurable interconnections. FPGAs have inferior metrics for area, performance and power, in comparison with application-specific integrated circuits (ASIC), but also have an infinitely smaller initial cost and production time. Moreover, FPGAs can be reprogrammed multiple times, with its reconfiguration made in seconds. For those reasons, FPGAs are largely used for ASIC prototyping and low-volume applications. Recently, FPGAs started to be employed in the optimization of specific tasks in data centers, with technology leaders making great efforts in hybrid solutions with ASICs and FPGAs [1], [2].

The FPGA implementation process inherited many techniques from the ASIC design flow, with high-level synthesis, logic synthesis and physical design. The use of well-established methods enabled the fast growing and wide-usage of FPGAs, but these algorithms generally have cost functions customized for cell-based designs, in which the simplest logic block has 2 inputs. Usual cost functions in logic synthesis are cubes in sum-of-product forms, literals in Boolean function expressions, or nodes and levels of and-inverter graphs (AIGs). On the other hand, look-up table (LUT) based FPGAs are composed of logic blocks with *k* inputs (typically 4 to 6), and each LUT can implement any logic function of up to *k* inputs. This difference leads to FPGA mappings with sub-optimal results, as observed in [3]. A study on this miscorrelation is presented in [4], showing that the reduction of nodes and levels in AIGs not necessarily translates to a FPGA mapping with fewer LUTs or less logic depth.

Several previous works on FPGA mapping are based on cut-enumeration, performing a covering of the subject graph with k-cuts [5]–[7]. FlowMap [5] was the first approach to guarantee minimal depth for a given structure, and the algorithms just improved ever since. These cut-based techniques vary on the algorithms, parameters, and cost functions used for the cut-enumeration and covering. Still, the quality of the solution heavily depends on the structure of the subject graph.

A second group of previous works rely on Binary Decision Diagrams (BDDs) as function representation to perform FPGA mapping [8]–[11]. The use of BDDs usually provides per se a good starting point for FPGA mapping, as the redundant variables are removed and the structure size is reduced. Also, BDDs enable the use of functional techniques, reducing the structural bias. However, the complexity of BDDs increases significantly with the number of variables, becoming computationally unfeasible for large designs. Thus, BDD-based methods often rely on *partial collapsing* of the subject graph, limiting the collapsing process based on the BDD size.

This work proposes to gather these two FPGA mapping strategies, with two main contributions:

- A functional decomposition which uses the support as the main cost function, and it is based on simple and known support-reducing techniques.
- An iterative collapsing-remapping approach, which aggressively tries to reduce the structural bias of the circuit and uses the FPGA mapping metrics as cost function.

On top of a fast and high-quality cut-based FPGA mapping algorithm [7] and AIG minimization methods [12]–[14], we propose an iterative collapsing strategy of the mapped FPGA network, exploiting the structural don't cares. A tree-based support-reducing decomposition method is applied, and a subject graph is generated, which has a structure more suitable for LUT-based FPGA mapping. Finally, the circuit parts are greedily selected based on the mapping result.

The paper is organized as follows. Some preliminary concepts are introduced in Section II. Section III presents the proposed support-reducing decomposition, and the iterative collapsing-remapping is depicted in Section IV. Section V provides the FPGA mapping results of the method proposed, and comparisons with academic tools and a commercial tool. Section VI concludes the paper.

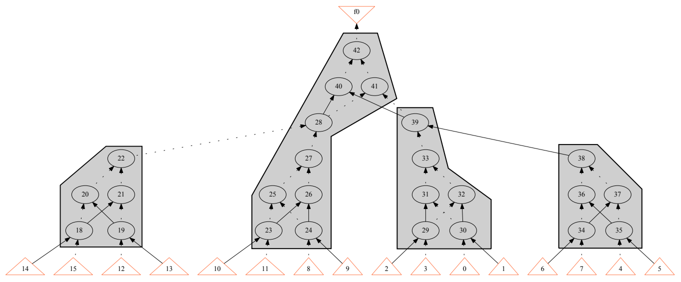

Fig. 1. Example of AIG with FPGA structural mapping (*t481* benchmark).

II. Preliminaries

A. Boolean Functions

An incompletely specified Boolean function (ISF) $F(X)$ of n Boolean variables is a mapping from an n-dimensional into a 1-dimensional Boolean space: $\{0,1\}^n \rightarrow \{0,1,-\}$, where '$-$' denotes a *don't care* value. The sub-domains of F that evaluate to '1', '0' and '$-$' are the ON-set, OFF-set and DC-set, respectively. F is a completely specified function if its DC-set is empty. The set $X = (x_1, x_2, \ldots, x_n)$ is denoted as the *support* of F, and $|F|$ denotes the size of X (n variables). The complement of F is denoted as $\overline{F}$. The logic operations AND, OR and XOR are denoted as '$\cdot$', '$+$', and '$\oplus$', respectively.

B. Cofactors and derivations

The positive (negative) *cofactor* operation of $F(X)$ with respect to the variable $x_i \in X$ consists of assigning x_i to one (zero) in $F(X)$, which can be represented as F_{x_i} ($F_{\overline{x_i}}$). A *cube-cofactor* consists of performing the cofactor operation recursively, e.g. assigning the variables $\{x_i, x_j\} \in X$ in $F(X)$ to $x_i = 0$ and $x_j = 1$, which can be denoted as $F_{\overline{x_i},x_j}$.

Cofactors can be used to extract information of F with respect to a variable in its support. Typical derivations are: the Boolean difference (1), the existential abstraction (2), the universal abstraction (3), the *Shannon expansion* (4) and the positive and negative *Davio expansion* (5).

$$\delta F/\delta x_i = F_{\overline{x_i}} \oplus F_{x_i} \tag{1}$$

$$\exists x_i F = F_{\overline{x_i}} + F_{x_i} \tag{2}$$

$$\forall x_i F = F_{\overline{x_i}} \cdot F_{x_i} \tag{3}$$

$$F = \overline{x_i} \cdot F_{\overline{x_i}} + x \cdot F_{x_i} \tag{4}$$

$$\begin{aligned} F &= x_i \cdot \delta F/\delta x_i \oplus F_{\overline{x_i}} \\ F &= \overline{x_i} \cdot \delta F/\delta x_i \oplus F_{x_i} \end{aligned} \tag{5}$$

C. Binary Decision Diagrams

A *Binary Decision Diagram* (BDD) is a well-known representation of Boolean functions [8], [9]. A BDD is a directed-acyclic graph (DAG) with two terminal nodes 0 and 1, and each nonterminal node represents a Boolean variable with two outgoing edges: the 0-edge and the 1-edge. A reduced-ordered BDD (ROBDD) is a minimized version of a BDD, in which the nonterminal nodes are organized in a fixed variable order, in such a way that the number of BDD nodes is reduced. In this work, ROBDDs are referred as BDDs.

D. And-Inverter Graphs

An *and-inverter graph* (AIG) is a DAG in which each node has either 0 incoming edges - the *primary inputs* (PI), or 2 incoming edges - the *AND nodes*, or 1 incoming edge - the *primary outputs* (PO). Each edge can be negated or not. Sequential elements are considered as PI/PO pairs. An AIG example is depicted in Fig. 1: the dotted lines are negated edges, the 25 circles are AND nodes, the 16 triangles at the bottom are PIs, and the triangle at the top is a PO.

The AIG representation has a high correlation with its derived circuit implementation, with the area correlated with the amount of AND nodes, and the delay proportional to the number of levels between PIs and POs. Balancing [15], rewriting [12], refactoring [13] and resubstitution [14] are AIG transformations used to reduce the nodes and levels of AIGs. Some commonly used scripts in ABC [16] are *dc2* and *compress2rs* [4], [17], which iterate these AIG transformations.

E. FPGA mapping

Technology mapping consists of transforming a technology-independent subject graph into a network of gates from a technology. In FPGAs, the technology consists of LUTs, which are blocks that can be configured to implement any logic function of up to k variables.

Structural hashing [13] is an operation which ensures that the AIG has only one AND node with the same incoming edges, considering permutation. In ABC [16], the subject graph is an structurally hashed AIG, and a structural FPGA mapping is performed on top of this structure [7]. Notice that the FPGA mapping may vary significantly for functionally equivalent but structurally different AIGs [18].

For example, the AIG depicted in Fig. 1 implements the functionality of the benchmark *t481*. By reading the initial description in ABC and applying AIG optimizations iteratively (100 iterations of the *dc2* script), the best result obtained has 66 LUTs. Instead of using the initial structure, the AIG of Fig. 1 can be derived with functional transformations, such as the decomposition method proposed in this paper. The FPGA mapping (k=6) obtained from the AIG of Fig. 1 is represented with the gray shapes (4 LUTs and 2 levels).

F. Collapsing

Collapsing [15] a circuit means to replace its logic by a Boolean network, with one node for each output, and each node implements the output function based solely on the primary inputs. Structural don't cares are removed when collapsing is performed, but also the logic sharing between outputs is lost. Notice that removing the shared logic may increase area, but also augments the possibilities of reducing delay. In order to obtain an AIG from a collapsed network, and therefore a subject graph for FPGA mapping, it is necessary to decompose the function of each node. For example, the decomposition can be performed via algebraic factorization [19] (*strash* command in ABC), or using the BDD structure, replacing each BDD node by a multiplexer (*muxes* command in ABC).

```
1:  function supportReduceDecomposition($F$, $DC$)
    Input: An ISF, with the ON-set ($F$) and the DC-set ($DC$)
    Output: A Boolean network (tree) that implements the $ISF$
2:      /* check for trivial cases (constants, variables) */
3:      /* the support size of $F$ is denoted as $|F|$ */
4:      if $|F| \leq 1$ then return $F$
5:      /* if DC-set is not empty, then minimize $F$ */
6:      if $DC \neq \emptyset$ then
7:          $F_{\min}$ = MINIMIZE($F$, $DC$)
8:          /* accept $F_{\min}$ if support is reduced or the same */
9:          if $|F_{\min}| \leq |F|$ then $F = F_{\min}$
10:     /* perform decomposition */
11:     $op, < F_1, ..., F_i, ..., F_n >$ = decomposeFunction($F$, $DC$)
12:     /* $op \in$ {AND, OR, XOR, MUX, AND-OR, AND-XOR} */
13:     for each function $F_i$ in $< F_1, ..., F_i, ..., F_n >$ do
14:         /* calculate satisfiability don't care as in [20] */
15:         $DC_i$ = $DC$ + calcSDC($op$, $i$, $< F_1, ..., F_i, ..., F_n >$)
16:         /* decompose $F_i$ recursively */
17:         $network$ = supportReduceDecomposition($F_i$, $DC_i$)
18:         $op$.connect($i$, $network$) /* connect network to gate input $i$ */
19:     return $op$
```

Fig. 2. Pseudo-code of the proposed support-reducing decomposition.

III. Support-reducing functional decomposition

This section presents the functional decomposition method proposed, which is based on support-reducing techniques. In this work, a decomposition of a function F is considered *support-reducing* if the derived functions have a support smaller than F. This definition differs from [21], which limits the term *support-reducing* to disjoint-support decompositions.

Fig. 2 depicts a pseudo-code of the algorithm. The method is technology-independent, i.e. it is agnostic to the FPGA technology. The input is an incompletely specified function (ISF), with its ON-set and DC-set in a functional representation, e.g. BDDs, truth tables. The result is a Boolean network that implements the ISF, composed of gates obtained from the decompositions tried: {*AND, OR, XOR, MUX, AND-OR, AND-XOR*}. Other gates could be derived for different techniques.

The trivial cases are checked at line 4. If the DC-set is not empty, then minimization is applied (line 7), updating F if the minimized function has a support smaller than or equal to F. The decomposition method described in Fig. 3 is called at line 11, which receives an ISF as input and returns a solution consisted of a decomposing gate *op* and a set of functions. If the solution is not disjoint-support, then the satisfiability don't care (SDC) conditions are calculated with calcSDC function (line 15), which is implemented as in [20].

Each derived function is decomposed recursively, generating a Boolean network (line 17). The network obtained is connected to the related input in the decomposing gate (*op*) at line 18. Notice that the resulting network is also a tree and no efforts are made to share logic in this method, leaving this task to AIG transformations in ABC (see details in Section IV).

Different support-reducing techniques are tried in the method of Fig. 3: using essential literals with the method described in Fig. 4 (lines 4-5), trying to remove one variable from the support using the method in Fig. 5 (line 9), and reduce two variables with the method shown in Fig. 6 (line 11). The existence of essential literals is checked first, and preferred to the other decomposition techniques. The remaining of the section details the cost function used to define the *best* solution in Fig. 3, as well as the support-reducing techniques applied.

```
1:  function decomposeFunction($F$, $DC$)
    Input: An ISF, with the ON-set ($F$) and the DC-set ($DC$)
    Output: A decomposing gate $op$, and
    the input functions $< F_1, ..., F_i, ..., F_n >$
2:      $Q = \emptyset$ /* priority queue of potential solutions */
3:      /* check for essential literals */
4:      decomposeEssentials($F$, 1, $Q$)
5:      decomposeEssentials($\overline{F}$, 0, $Q$)
6:      /* if essential literals found, return */
7:      if $Q \neq \emptyset$ then return best solution $\in Q$
8:      /* check one-variable decompositions */
9:      decomposeOneVariable($F$, $DC$, $Q$)
10:     /* check two-variable decompositions */
11:     decomposeTwoVariables($F$, $DC$, $Q$)
12:     return best solution $\in Q$
```

Fig. 3. Pseudo-code for an step of the support-reducing decomposition.

```
1:  procedure decomposeEssentials($F$, $P$, $Q$)
    Input: Boolean function $F$, polarity $P$, priority queue $Q$
    Post: solutions added to priority queue $Q$
2:      $E =< l_1, l_2, ..., l_i, ..., l_n >$ /* set of $n$ essential literals of $F$ */
3:      if $E == \emptyset$ then return
4:      $H = F_{l_1, l_2, l_i, l_n}$ /* cube-cofactor of $F$ w.r.t. $E$ */
5:      if $H \neq 1$ then
6:          $G = (l_1 \cdot l_2 \cdot l_i \cdot l_n)$ /* AND of all essential literals */
7:          if $P \neq 0$ then $Q.add(G \cdot H)$
8:          else $Q.add(\overline{G} + \overline{H})$
9:      else
10:         $G_1 = (l_1 \cdot l_2)$ /* AND of essential literals 1 to $\frac{n}{2}$ */
11:         $G_2 = (l_i \cdot l_n)$ /* AND of essential literals $\frac{n}{2}$+1 to $n$ */
12:         if $P \neq 0$ then $Q.add(G_1 \cdot G_2)$
13:         else $Q.add(\overline{G_1} + \overline{G_2})$
```

Fig. 4. Pseudo-code for decomposition using essential literals.

A. Cost function

The pseudo-code in Fig. 3 returns the *best* solution from a priority queue (lines 7 and 12). In this work, the best solution is the one that obtains the set of derived functions with the smallest sum of support sizes. Moreover, a solution is discarded if one of the derived functions has the same support as F. Additionally, if there is more than one solution with the smallest sum of support sizes, then the following costs are considered, in this order:

1) The sum of the squares of the BDD sizes [22], targeting a balanced solution, which favors delay reduction.
2) The gate implementation cost in CMOS transistors, e.g. an AND gate costs less than a MUX or an XOR.

As BDDs are the function representation of choice, the costs exploit its structure to guide the decomposition, but similar costs could be derived for different representations.

B. Essential literals

Fig. 4 describes the decomposition method using *essential literals*, i.e. literals that are common to all prime implicants. For example, given $F(X)$ and $\{a, b, c\} \in X$, if $\{a, b, \bar{c}\}$ are essential literals of $F(X)$, then F can be rewritten as $F(X) = (a \cdot b \cdot \bar{c}) \cdot F_{a,b,\bar{c}}$. Similarly, given $G(X) = \overline{F(X)}$ and $\{x, y, z\} \in X$, if $\{\bar{x}, y, z\}$ are essential literals of $G(X)$, then F can be decomposed as $F(X) = \overline{(\bar{x} \cdot y \cdot z)} + \overline{G_{\bar{x},y,z}}$.

The decomposition with the essential literals of F is checked at line 7, and the one for $\overline{F}$ at line 8. If the function is solely composed of essential literals, i.e. the cube-cofactor w.r.t. to the essential literals is the constant 1 (F is a cube), then a balanced decomposition is performed (lines 12-13).

```
 1: procedure DECOMPOSEONEVARIABLE($F$, $DC$, $Q$)
    Input: The ON-set ($F$), DC-set ($DC$), and priority queue $Q$
    Post: solutions added to priority queue $Q$
 2:     for each variable $x_i \in$ support of $F$ do
 3:         if $\delta F/\delta x_i == 1$ then
 4:             $Q.add(x_i \oplus F_{\overline{x_i}})$
 5:             $Q.add(\overline{x_i} \oplus F_{x_i})$
 6:         else
 7:             if $\exists x_i F == F_{\overline{x_i}}$ then
 8:                 $Q.add((\overline{x_i} \cdot F_{\overline{x_i}}) + F_{x_i})$  /* AND-OR */
 9:             else if $\exists x_i F == F_{x_i}$ then
10:                 $Q.add((x_i \cdot F_{x_i}) + F_{\overline{x_i}})$  /* AND-OR */
11:             else /* full Davio and Shannon expansions */
12:                 $Q.add((x_i \cdot \delta F/\delta x_i) \oplus F_{\overline{x_i}})$ /* AND-XOR */
13:                 $Q.add((\overline{x_i} \cdot \delta F/\delta x_i) \oplus F_{x_i})$ /* AND-XOR */
14:                 $Q.add(\overline{x_i} \cdot F_{\overline{x_i}} + x \cdot F_{x_i})$  /* MUX */
15:             /* one-variable abstraction-based decompositions */
16:             if $\exists x_i F \neq 1$ then $G$ = $\exists x_i F$
17:                 $H$ = MINIMIZE($F$, $\overline{G} + DC$)
18:                 $Q.add(G \cdot H)$
19:             if $\forall x_i F \neq 0$ then $G$ = $\forall x_i F$
20:                 $H$ = MINIMIZE($F$, $G + DC$)
21:                 $Q.add(G + H)$
```

Fig. 5. Pseudo-code for one-variable decompositions.

C. One-variable decompositions

The basic one-variable support-reducing decompositions are given by the Shannon expansion (4), and the Davio expansions (5). These methods isolate one variable, therefore reducing the support size of the derived functions in at least one. Simplifications of these expansions can be obtained given specific conditions, as shown in (6). Essential literals cover the cases in which one of the cofactors is a constant.

$$\begin{aligned} F &= x_i \oplus F_{\overline{x_i}}, && \text{if } \delta F/\delta x_i = 1 \\ F &= \overline{x_i} \oplus F_{x_i}, && \text{if } \delta F/\delta x_i = 1 \\ F &= \overline{x_i} \cdot F_{\overline{x_i}} + F_{x_i}, && \text{if } \exists x_i F = F_{\overline{x_i}} \\ F &= x_i \cdot F_{x_i} + F_{\overline{x_i}}, && \text{if } \exists x_i F = F_{x_i} \end{aligned} \tag{6}$$

The Davio and Shannon expansions are added to the queue in the method described in Fig. 5 (lines 12-14). The simplifications listed in (6) are also checked (lines 4-5, 8 and 10) and preferred to the full Davio and Shannon expansions.

D. Two-variable decompositions

In [23], it is proposed the use of simple cofactor tests in order to perform disjoint-support decompositions (DSD). The cofactor tests and decompositions for AND and XOR are described in (7), given $F(X)$ and $\{x, y\} \in X$. These tests are performed in the method of Fig. 6 (lines 4-10, and 17).

$$\begin{aligned} F &= \overline{x \cdot y} \cdot F_{\bar{x}} + x \cdot y \cdot F_{x,y}, && \text{if } F_{\bar{x}} = F_{\bar{y}} \\ F &= \overline{x \cdot \bar{y}} \cdot F_{\bar{x}} + x \cdot \bar{y} \cdot F_{x,\bar{y}}, && \text{if } F_{\bar{x}} = F_{y} \\ F &= \overline{\bar{x} \cdot y} \cdot F_{x} + \bar{x} \cdot y \cdot F_{\bar{x},y}, && \text{if } F_{x} = F_{\bar{y}} \\ F &= \overline{\bar{x} \cdot \bar{y}} \cdot F_{x} + \bar{x} \cdot \bar{y} \cdot F_{\bar{x},\bar{y}}, && \text{if } F_{x} = F_{y} \\ F &= ((x \oplus y) \cdot \delta F/\delta x) \oplus F_{\bar{x},\bar{y}}, && \text{if } \delta F/\delta x = \delta F/\delta y \end{aligned} \tag{7}$$

If one of the cube-cofactors in (7) is a constant, then simplifications can be derived, which are checked in Fig. 6 (lines 14-15, and 19-21). If no simplification is possible, then a MUX gate is defined for the AND decomposition (line 16), and an AND-XOR gate for the XOR decomposition (line 22).

```
 1: procedure DECOMPOSETWOVARIABLES($F$, $DC$, $Q$)
    Input: The ON-set ($F$), DC-set ($DC$), and priority queue $Q$
    Post: solutions added to priority queue $Q$
 2:     for each pair of variables $x_i, x_j \in$ support of $F$ do
 3:         $C$ = true  /* detects an AND DSD condition */
 4:         if $F_{\overline{x_i}} == F_{\overline{x_j}}$ then
 5:             $S$ = $(x_i \cdot x_j)$; $G$ = $F_{x_i,x_j}$; $H$ = $F_{\overline{x_i}}$
 6:         else if $F_{\overline{x_i}} == F_{x_j}$ then
 7:             $S$ = $(x_i \cdot \overline{x_j})$; $G$ = $F_{x_i,\overline{x_j}}$; $H$ = $F_{\overline{x_i}}$
 8:         else if $F_{x_i} == F_{\overline{x_j}}$ then
 9:             $S$ = $(\overline{x_i} \cdot x_j)$; $G$ = $F_{\overline{x_i},x_j}$; $H$ = $F_{x_i}$
10:         else if $F_{x_i} == F_{x_j}$ then
11:             $S$ = $(\overline{x_i} \cdot \overline{x_j})$; $G$ = $F_{\overline{x_i},\overline{x_j}}$; $H$ = $F_{x_i}$
12:         else $C$ = false
13:         if $C$ then
14:             if $G == 0$ then $Q.add(\overline{S} \cdot H)$
15:             else if $G == 1$ then $Q.add(S + H)$
16:             else $Q.add(\overline{S} \cdot H + S \cdot G)$  /* MUX */
17:         else if $\delta F/\delta x_i == \delta F/\delta x_j$ then
18:             $S = (x_i \oplus x_j)$, $G = F_{\bar{x},\bar{y}}$, $H = \delta F/\delta x_i$
19:             if $G == 0$ then $Q.add(S \cdot H)$
20:             else if $G == 1$ then $Q.add(\overline{S} + \overline{H})$
21:             else if $H == 1$ then $Q.add(S \oplus G)$
22:             else $Q.add((S \cdot H) \oplus G)$  /* AND-XOR */
23:         else /* two-variable abstraction-based decompositions */
24:             if $\exists x_i x_j F \neq 1$ then $G$ = $\exists x_i x_j F$
25:                 $H$ = MINIMIZE($F$, $\overline{G} + DC$)
26:                 $Q.add(G \cdot H)$
27:             if $\forall x_i x_j F \neq 0$ then $G$ = $\forall x_i x_j F$
28:                 $H$ = MINIMIZE($F$, $G + DC$)
29:                 $Q.add(G + H)$
```

Fig. 6. Pseudo-code for two-variable decompositions.

E. Abstraction-based decompositions

This work introduces two decompositions based on the existential and universal abstractions. As explained in [24],

$$\forall x_i F \leq F \leq \exists x_i F. \tag{8}$$

Thus, the existential abstraction $\exists x_i F$ implies an AND bi-decomposition, and the universal abstraction $\forall x_i F$ implies an OR bi-decomposition. These decompositions guarantee the support reduction for at least one of the derived functions.

The condition to define H in the AND bi-decomposition $F = G \cdot H$ is $F \leq H \leq F + \overline{G}$, given F and G. Considering $G = \exists x_i F$, then $F \leq H \leq F + \overline{\exists x_i F}$. Similarly, the condition to define H in the OR bi-decomposition $F = G + H$ is $F \cdot \overline{G} \leq H \leq F$, given F and G. Considering $G = \forall x_i F$, then $F \cdot \overline{\forall x_i F} \leq H \leq F$. These conditions are used to define H via don't care minimization. Additionally, the minimization can take advantage of the satisfiability don't cares generated by previous decompositions. The minimization can be performed by any method that accepts an ISF, such as Espresso [25], or BDD minimization [26], [27]. In this work, the abstraction-based decompositions are applied to one variable (lines 16-21 in Fig. 5) and two variables (lines 24-29 in Fig. 6).

IV. ITERATIVE COLLAPSING AND REMAPPING

This section presents an alternative and aggressive partial collapsing approach. The idea is to iteratively collapse the subject network, selecting the best FPGA mapping for each circuit part. A pseudo-code detailing the proposed approach is shown in Fig. 8, and an overview of the method is depicted in Fig. 7. The method input is an FPGA mapping (R_0), the number of LUT inputs k, and a cost function COST, e.g. area or delay. The result is an optimized implementation of R_0.

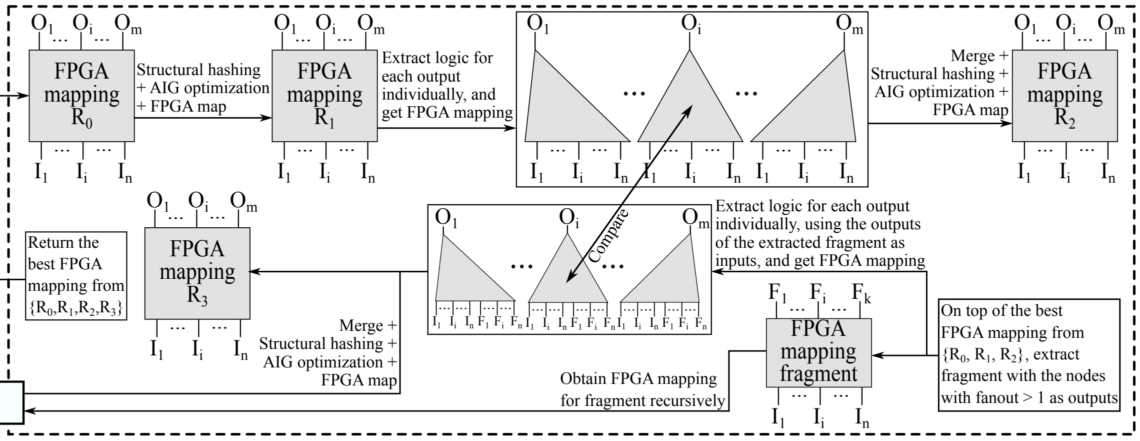
FPGA mapping R_0
Structural hashing + AIG optimization + FPGA map
FPGA mapping R_1
Extract logic for each output individually, and get FPGA mapping
Merge + Structural hashing + AIG optimization + FPGA map
FPGA mapping R_2
Compare
Extract logic for each output individually, using the outputs of the extracted fragment as inputs, and get FPGA mapping
On top of the best FPGA mapping from $\{R_0, R_1, R_2\}$, extract fragment with the nodes with fanout > 1 as outputs
FPGA mapping fragment
Obtain FPGA mapping for fragment recursively
Merge + Structural hashing + AIG optimization + FPGA map
FPGA mapping R_3
Return the best FPGA mapping from $\{R_0,R_1,R_2,R_3\}$
O_1 O_i O_m
I_1 I_i I_n
F_1 F_i F_k

Fig. 7. The iterative collapsing-remapping approach.

The first step is a remapping of R_0 using the function FPGAMAP (line 5). This function performs structural hashing, 10 iterations of AIG optimizations scripts, and an FPGA mapping with structural choices [28] for each structure generated, returning the best mapping for the cost function COST. The number of iterations can be incremented to try to get better results, or it can be reduced for a lower runtime.

The second step is the collapsing of each output, which is performed in function COLLAPSEFPGAMAP (lines 16 and 31). Collapsing is a computationally complex process that may be unfeasible for complex networks, so a time-out is set to avoid indefinite execution. If collapsing is successful, then the method presented in Section III is applied, generating a new structure. If collapsing is not successful, then the single output network extracted is the only structure considered. On top of each of these structures, structural hashing is performed, followed by a single execution of AIG optimization scripts, with an FPGA mapping with structural choices performed for each different structure, returning the best mapping result for each output. FPGA mapping R_2 is generated by merging the output networks obtained (line 19) using structural hashing, followed by the function FPGAMAP (line 21).

The third step consists of extracting a *fragment network* from the best FPGA mapping produced so far, in which its outputs are the nodes with multiple fanout identified in topological order. These nodes are transformed to primary inputs from the outputs perspective, and the collapsing-remapping process is repeated for each output. The implementation for the shared *fragment network* is obtained recursively, until the number of levels is 1 (line 8), returning the best FPGA mapping for each recursive call. FPGA mapping R_3 is generated by merging the output networks and the *fragment* result (line 40) using structural hashing, followed by FPGAMAP (line 42).

```
 1: function ITERATIVECOLLAPSE($R_0$, $k$, COST)
    Input: FPGA mapping $R_0$, LUT size $k$, cost function COST
    Output: An optimized FPGA mapping guided by COST
 2:     BestR = $R_0$ /* initialize the best result to be returned */
 3:     /* obtain $R_1$ by remapping input $R_0$ */
 4:     /* run structural hashing, AIG optimizations, FPGA mapping */
 5:     $R_1$ = FPGAMAP($R_0$, $k$, COST)
 6:     if COST($R_1$) $<$ COST(BestR) then BestR = $R_1$
 7:     /* if the number of levels is 1, return */
 8:     if number of levels of BestR == 1 then return BestR
 9:     /* obtain $R_2$ by remapping outputs individually */
10:     outputNetworks = $\emptyset$ /* best implementation for each output */
11:     /* implement each output individually */
12:     for each output $i$ in BestR do
13:         /* extract single output network */
14:         Ntk_$O_i$ = EXTRACTOUTPUTTOINPUTS(BestR, $i$)
15:         /* collapse, decompose, optimize, FPGA mapping */
16:         $O_i$ = COLLAPSEFPGAMAP(Ntk_$O_i$, $k$, COST)
17:         outputNetworks.insert($O_i$, $i$)
18:     /* merge output networks using structural hashing */
19:     Ntk_$R_2$ = MERGENETWORKS(outputNetworks)
20:     /* run AIG optimizations, FPGA mapping */
21:     $R_2$ = FPGAMAP(Ntk_$R_2$, $k$, COST)
22:     if COST($R_2$) $<$ COST(BestR) then BestR = $R_2$
23:     /* obtain $R_3$ by remapping outputs with a shared fragment */
24:     tempNtk = MULTIPLEFANOUTTOPI(BestR)
25:     sharedNodes = $\emptyset$ /* set of shared nodes used */
26:     /* implement each output individually */
27:     for each output $i$ in tempNtk do
28:         /* extract single output network */
29:         Ntk_$O_i$ = EXTRACTOUTPUTTOINPUTS(tempNtk, $i$)
30:         /* collapse, decompose, optimize, FPGA mapping */
31:         $O_i$ = COLLAPSEFPGAMAP(Ntk_$O_i$, $k$, COST)
32:         if COST($O_i$) $<$ COST(outputNetworks[$i$]) then
33:             outputNetworks.insert($O_i$, $i$)
34:             sharedNodes.insert(inputs of $O_i$)
35:     if sharedNodes $\neq \emptyset$ then
36:         /* get shared fragment with sharedNodes as outputs */
37:         fragment = GETSHAREDFRAGMENT(BestR, sharedNodes)
38:         $R_f$ = ITERATIVECOLLAPSE(fragment, $k$, COST)
39:         /* merge output networks and fragment using structural hashing */
40:         Ntk_$R_3$ = MERGENETWORKS(outputNetworks, $R_f$)
41:         /* run AIG optimizations, FPGA mapping */
42:         $R_3$ = FPGAMAP(Ntk_$R_3$, $k$, COST)
43:         if COST($R_3$) $<$ COST(BestR) then BestR = $R_3$
44:     return BestR
```

Fig. 8. Pseudo-code of the iterative collapsing-remapping approach.

V. Experimental results

The support-reducing functional decomposition and the iterative collapsing-remapping are implemented in C++. BDDs are the functional representation of choice for the functional decomposition, and the CUDD BDD package [27] is used. All results passed formal verification with ABC command *cec*.

The FPGA mapping based in priority cuts [7] and choices [28] implemented in ABC [16] is the one used in the iterative collapsing-remapping approach. Delay-oriented FPGA mapping with ABC is obtained by the command '*if -C 12 -K k*', which targets primarily delay, with a configuration of at most 12 priority cuts per node [17].

TABLE I
FPGA MAPPING COMPARISON WITH BDD-BASED APPROACHES ($k = 5$).

Circuit	BoolMap [8] (delay)		BDS-pga [9] (delay)		ABC (delay)		SR-map (delay) + ABC (delay)		
	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	Time(s)
5xp1	**13**	**2**	15	2	21	3	14	2	3
9sym	**7**	**3**	**7**	**3**	60	4	8	3	2
9symml	**7**	**3**	**7**	**3**	58	4	8	3	2
alu2	43	4	41	4	117	7	**35**	**4**	13
alu4	268	7	190	7	219	9	**102**	**4**	26
apex6	188	4	186	4	171	4	**169**	**3**	26
apex7	78	3	71	3	61	3	**54**	**3**	8
b9	41	3	40	3	33	3	**38**	**2**	3
C1355	98	5	**65**	**4**	66	4	66	4	157
C1908	137	7	119	7	95	6	**86**	**6**	217
C499	102	4	**64**	**4**	66	4	66	4	162
C5315	672	9	447	7	365	7	**374**	**6**	217
C880	134	8	108	8	87	7	**92**	**6**	55
clip	**15**	**2**	30	4	71	4	19	3	8
count	**42**	**2**	26	5	36	3	33	3	5
des	**594**	**3**	909	4	623	4	568	4	328
duke2	192	5	169	7	141	4	**120**	**4**	26
misex1	15	2	14	2	15	2	**11**	**2**	2
rd84	**10**	**2**	13	3	109	5	13	3	11
rot	228	6	218	9	203	6	**215**	**5**	43
t481	5	3	**5**	**2**	148	6	**5**	**2**	2
vg2	30	4	**12**	**3**	27	3	21	3	6
z4ml	**5**	**2**	**5**	**2**	**5**	**2**	5	2	1
Geomean	49.63	3.60	44.95	3.91	74.81	4.20	**40.59**	**3.31**	57.52
Ratio	1.00	1.00	0.91	1.09	1.51	1.17	0.82	0.92	-

Alternatively, area-oriented FPGA mapping is applied by setting the parameter '*-a*', with the command '*if -a -C 12 -K k*'. The number of LUT inputs varies for the different sets of benchmarks. Also, structural bias is further reduced by identifying structural choices [28], which is performed by running two FPGA mappings, with the commands '*&synch2*' and '*&dch*' before each mapping, and selecting the best result between the two.

The proposed approach, named as **S**upport-**R**educing Re**map**ping tool (**SR-map**), attempts to optimize a given FPGA mapping. In the experiments presented in this section, the *input* FPGA mapping (R_0) is either the one obtained with ABC, or the best known result for the EPFL benchmarks [29].

The FPGA mappings obtained are greedily selected based on the cost function. In this work, two cost functions are analyzed: logic levels and LUT count. If the objective is reducing delay, then SR-map greedily selects the circuit parts with fewer logic levels, using LUT count as a tie breaker. Alternatively, if the goal is to minimize area, LUT count is the main cost function and logic levels is the tie breaker.

A. BDD-based FPGA mapping tools

This section compares SR-map results with the FPGA mappings reported by the BDD-based tools BoolMap [8] and BDS-pga [9], and the ones obtained with ABC. The FPGA mappings of BoolMap and BDS-pga refer to the best results for delay reported in [8] and [9], which are presented in Table I, with FPGA mapping to LUTs with k=5. The bold numbers in Table I highlight the best results for delay. BDS-pga achieves an area reduction of 9% in comparison with BoolMap, at the expense of increasing delay in 9%.

The ABC results are obtained using the same input structure as the BDD-based tools, applying structural hashing, 10 iterations of AIG optimizations ($compress2rs$ and $dc2$ scripts), and mapping for FPGA with the command '*if -C 12 -K 5*', identifying structural choices with commands '*&synch2*' and '*&dch*'. For this set of benchmarks and configuration, ABC produces worse results than the BDD-based tools, with mappings 51% larger in area and 17% larger in delay, compared to BoolMap. The difference in LUTs is larger than 90% for benchmarks *rd84* and *t481*, and the reason for this behavior lies on the nature of each approach. BoolMap and BDS-pga perform functional transformations using BDDs, whereas ABC performs an structural mapping on top of an AIG, in which its nodes and levels are iteratively minimized.

SR-map is able to boost ABC structural bias reduction, delivering a result that outperforms BoolMap [8], even using the networks generated by ABC as starting point. This work improves BoolMap results in 18% for area and 8% for delay, with the best delay result for 12 of the 22 benchmarks.

B. 20 largest MCNC benchmarks

The BDD-based methods deliver reasonably good results, but are not scalable for large designs. As this work relies only partially on BDDs, it is possible to apply it to larger circuits. This section presents results for the 20 largest MCNC benchmarks, comparing the methods proposed in this work with a commercial tool and ABC.

Table II presents the results with FPGA mappings to LUTs with k=6. The synthesis in the commercial tool is configured to avoid the use of multiplexers and merging of LUTs, delivering results comparable to the other tools. The reported runtime considers only the logic synthesis and optimization steps. In Table II, the bold numbers in '*Levels*' highlight best delay results, whereas the bold numbers in '*LUTs*' underline the best results considering area.

1) FPGA mapping for delay: The ABC results for delay are obtained using the input structure, applying structural hashing and iterative AIG optimizations, and mapping for FPGA with the command '*if -C 12 -K 6*', using commands '*&synch2*' and '*&dch*' to identify structural choices. This process produces a result 99% larger in area and 5% larger in delay than the commercial tool.

Using ABC delay-oriented mapping and delay as cost function for the selection of the circuit parts, SR-map produces a result with 27% fewer logic levels and 10% fewer LUTs than the commercial tool. Also, an area reduction of 16% with a delay reduction of 12% is achieved when area minimization is defined as the cost function.

2) FPGA mapping for area: The ABC results for area are obtained using the command '*if -a -C 12 -K 6*', also identifying structural choices. Notice that there is an area recovery post-process in ABC delay-oriented mapping, but area-oriented mapping does not try to improve delay. This setup produces results with fewer LUTs, still 83% larger in area than the commercial tool, but increases significantly the delay results, almost doubling logic levels.

TABLE II
FPGA MAPPING COMPARISON FOR THE 20 LARGEST MCNC BENCHMARKS ($k = 6$).

								Iterative collapsing and remapping											
Circuit	Commercial tool			ABC (delay)		ABC (area)		Factor (delay) + ABC (delay)		Factor (area) + ABC (delay)		SR-map (delay) + ABC (delay)			SR-map (area) + ABC (delay)			SR-map (area) + ABC (area)	
	LUTs	Lev.	Time(s)	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	Time(s)	LUTs	Lev.	Time(s)	LUTs	Lev.
alu4	320	5	230	456	5	415	10	183	5	186	5	63	**3**	24	58	4	21	**56**	5
apex2	302	13	276	507	6	408	11	40	4	40	4	35	**3**	17	34	4	14	**30**	7
apex4	192	3	290	558	5	529	10	390	4	390	4	155	3	203	155	3	212	**153**	**3**
bigkey	569	3	313	577	3	577	3	577	3	577	3	685	**2**	257	**491**	3	191	577	3
clma	**180**	5	438	2614	8	2221	18	200	4	203	4	203	4	75	190	**4**	84	186	8
des	**436**	4	345	447	4	457	7	447	4	446	4	513	**3**	194	445	4	222	449	5
diffeq	**472**	8	348	559	7	510	13	559	7	532	8	533	**7**	241	527	8	262	502	14
dsip	**690**	3	338	871	3	871	3	869	**2**	869	**2**	869	**2**	190	869	**2**	224	869	**2**
elliptic	**115**	**5**	313	315	6	297	11	315	6	315	6	316	5	36	311	6	51	291	11
ex1010	210	3	335	572	5	550	10	453	4	432	5	208	**3**	107	**207**	4	180	**207**	4
ex5p	100	2	252	326	4	301	9	91	2	85	3	86	2	12	**82**	**2**	16	82	5
frisc	1694	13	292	1725	12	1698	26	1886	10	1692	13	1857	**10**	654	1689	13	853	**1637**	25
i10	557	9	285	535	8	500	22	627	7	522	9	537	**7**	329	505	9	317	**481**	24
misex3	197	5	240	284	5	234	9	205	4	193	4	117	4	37	102	**4**	35	**94**	5
pdc	155	4	286	1385	6	1143	14	154	4	148	4	157	**3**	70	**142**	4	67	144	6
s38417	**1458**	7	437	2557	6	2443	11	2460	6	2458	7	2450	**6**	1304	2396	7	1035	2369	12
s38584	**1946**	8	432	2287	6	2255	12	2463	5	2212	6	2224	**5**	1246	2229	5	729	2198	12
seq	531	7	247	583	5	520	10	560	4	486	5	527	**4**	307	459	5	114	**420**	10
spla	157	4	269	1350	6	1128	15	145	4	137	4	156	**3**	138	135	4	62	**121**	6
tseng	656	8	269	651	6	631	13	647	6	635	8	634	**6**	385	636	8	265	**629**	12
Geomean	374.30	5.24	306.1	744.07	5.51	685.86	10.56	400.08	4.43	383.36	4.92	335.79	**3.84**	141.1	312.67	4.61	128.9	**304.62**	7.21
Ratio	1.00	1.00	-	1.99	1.05	1.83	2.01	1.07	0.84	1.02	0.94	0.90	**0.73**	-	0.84	0.88	-	**0.81**	1.38

TABLE III
BEST KNOWN RESULTS FOR EPFL BENCHMARKS 2017 ($k = 6$).

Circuit	Best EPFL (delay)		SR-map (delay) + ABC (delay)		
	LUTs	Levels	LUTs	Levels	Time(s)
arbiter	2884	5	2243	5	729
cavlc	115	4	75	3	25
dec	270	2	264	2	11
int2float	41	3	31	3	5
i2c	244	3	242	3	30
mem_ctrl	2490	7	2484	6	792
max	882	10	857	10	313
multiplier	8215	28	6543	28	26012
priority	157	4	152	4	15
router	57	4	54	4	5
sin	1801	30	3546	28	12779
voter	1469	12	1450	12	4601
Circuit	Best EPFL (area)		SR-map (area) + ABC (area)		
	LUTs	Levels	LUTs	Levels	Time(s)
cavlc	101	6	72	4	25
dec	270	2	264	2	11
int2float	28	6	27	6	5

Using ABC area-oriented mapping and area as cost function, SR-map obtains a result with 19% fewer LUTs than the commercial tool, but with 38% more logic levels. The results are 3% smaller in area than using ABC delay-oriented mapping, but with much worse delay results, as this is disregarded in the area-oriented mapping.

3) Support-reducing decomposition: The remapping approach proposed in Section IV can be applied regardless of the support-reducing decomposition presented in Section III. For example, the collapsed functions can be decomposed using algebraic factorization [19], instead of the decomposition method proposed. The results for the iterative collapsing-remapping using factorization (obtained with the ABC command *strash*) instead of the support-reducing decomposition are also presented in Table II, denoted as '*Factor*'.

For some benchmarks, e.g. *apex2*, *clma*, *ex5p*, the iterative collapsing-remapping produces similar results both for the factorization and the decomposition. However, considering the full set of benchmarks, the results obtained using the support-reducing decomposition are considerably better than the ones using factorization, both for area and delay. Regarding the methods analyzed, SR-map obtains the best delay result for 19 of the 20, and the best area for 13 of the 20 benchmarks.

C. EPFL benchmarks

The EPFL benchmarks [29] are a set of 20 designs, 10 arithmetic and 10 random/control circuits. Since 2015, the best known FPGA mapping results (with k=6) for delay and for area are recorded. Consequently, these benchmarks have highly optimized results, which are very difficult to improve. For example, the commercial tool used in this work is not able to improve any of the EPFL results, as it provides FPGA mappings with more balanced results in area and delay.

The proposed method is able to update 12 of the best known results for delay (and 3 for area), as depicted in Table III. The most remarkable results are: *cavlc*, with a reduction of 25% in delay plus 35% in area; *int2float*, reducing LUT count in 25%; and the *multiplier*, with an area reduction of 20%. Note that the area of the *sin* benchmark is increased significantly for a reduction of 2 logic levels. This behavior is expected, as the delay is the cost function in this case. Therefore, SR-map greedily selects the circuit parts with lowest logic levels, considering LUT count only as a tie breaker.

D. Runtime analysis

Collapsing the network is a bottleneck for the method, as failed collapsing processes represent most of the execution time. Also, as a logic fragment is extracted considering all outputs, the number of levels in the FPGA mapping is also a limiting factor regarding performance. The ABC runtime is a fraction of the proposed approach, as it is repeatedly used. Still, the average execution time is comparable with the one for the commercial tool.

VI. Conclusions

This paper proposes a support-reducing functional decomposition method to produce a subject graph with a structure more suitable to LUT-based FPGA technology mapping. An iterative collapsing-remapping approach is also proposed, trying to reduce the structural bias of the circuit, while using the actual FPGA mapping result as cost function.

The experiments show promising results. The proposed method improves the results of a commercial tool for MCNC benchmarks, with gains of 27% in delay plus 10% in area when targeting delay, and 16% in area plus 12% in delay with area as cost function. Moreover, 12 of the best known results for delay (and 3 for area) of the EPFL benchmarks are updated.

A. Future work

As future work, some directions can be explored. Additional support-reducing techniques could be incorporated, such as [30] and [31], which propose scalable bi-decomposition methods. Regarding the remapping method, the propagation of the don't care conditions could potentially improve the results obtained. Also, keeping track of the critical paths may allow area reduction while obtaining similar delay results.

Acknowledgment

This work was performed with the support of CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil, and has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2017-86727-C2-1-R, and the Generalitat de Catalunya (2017 SGR 786).

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[2] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. of the Design Automation Conference*. ACM, 2016, p. 109.

[3] J. Cong and K. Minkovich, "Optimality study of logic synthesis for LUT-based FPGAs," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 230–239, 2007.

[4] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for LUT-based technology mapping," in *Proc. of the International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 147–156.

[5] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[6] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2004, pp. 752–759.

[7] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2007, pp. 354–361.

[8] C. Legl, B. Wurth, and K. Eckl, "A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs," in *Proc. of the Design Automation Conference*. ACM, 1996, pp. 730–733.

[9] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 501–525, 2002.

[10] L. Cheng, D. Chen, and M. D. Wong, "DDBDD: Delay-driven BDD synthesis for FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1203–1213, 2008.

[11] M. Kubica, A. Opara, and D. Kania, "Logic synthesis for FPGAs based on cutting of BDD," *Microprocessors and Microsystems*, vol. 52, pp. 173–187, 2017.

[12] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. of the Design Automation Conference*. ACM, 2006, pp. 532–535.

[13] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. of the International Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[14] R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis," in *Proc. of the International Workshop on Logic and Synthesis*, 2007, pp. 358–364.

[15] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[16] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. Springer, 2010, pp. 24–40.

[17] A. Mishchenko, R. Brayton, J. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, p. 34, 2011.

[18] P. Fišer, J. Schmidt, and J. Balcárek, "Sources of bias in EDA tools and its influence," in *Proc. of the International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2014, pp. 258–261.

[19] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. of International Symposium on Circuits and Systems*, 1982, pp. 49–54.

[20] H. Savoj, M. J. Silva, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Boolean matching in logic synthesis," in *Proc. of the conference on European design automation*, 1992, pp. 168–174.

[21] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," in *Proc. of Design, Automation and Test in Europe*. ACM, 2000, pp. 208–215.

[22] D. Bañeres, J. Cortadella, and M. Kishinevsky, "Timing-driven N-way decomposition," in *Proc. of the Great Lakes Symposium on VLSI*, 2009, pp. 363–368.

[23] V. Callegaro, F. S. Marranghello, M. G. Martins, R. P. Ribas, and A. I. Reis, "Bottom-up disjoint-support decomposition based on cofactor and Boolean difference analysis," in *Proc. of the IEEE International Conference on Computer Design*. IEEE, 2015, pp. 680–687.

[24] F. M. Brown, *Boolean reasoning: the logic of Boolean equations*. Springer Science & Business Media, 2012.

[25] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984, vol. 2.

[26] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, "Safe BDD minimization using don't cares," in *Proc. of the Design Automation Conference*. ACM, 1997, pp. 208–213.

[27] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0," *University of Colorado at Boulder*, 2015.

[28] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[29] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. of the International Workshop on Logic & Synthesis*, 2015.

[30] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large Boolean functions via interpolation and satisfiability solving," in *Proc. of the 45th annual Design Automation Conference*. ACM, 2008, pp. 636–641.

[31] M. Choudhury and K. Mohanram, "Bi-decomposition of large Boolean functions using blocking edge graphs," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2010, pp. 586–591.