Area-Oriented Resubstitution For Networks of Look-Up Tables

Andrea Costamagna, Alessandro Tempia Calvino, *Member, IEEE*, Alan Mishchenko, *Senior Member, IEEE*, and Giovanni De Micheli *Life Fellow, IEEE*

***Abstract*—This paper addresses the challenge of reducing the number of nodes in Look-Up Table (LUT) networks with two significant applications. First, Field-Programmable Gate Arrays (FPGAs) can be modelled as networks of LUTs, and minimizing the node count is imperative to meet resource constraints. Second, in area-oriented design space exploration for standard-cell designs, collapsing a circuit into a LUT network, restructuring it, and later remapping to the original representation helps escape local minima. Thus, the development of algorithms for optimizing and restructuring LUT networks holds considerable promise for area-oriented optimization. *Substitution* (also called resubstitution) is a powerful logic minimization method that can identify non-local logic dependencies and exploit them for logic minimization. State-of-the-art substitution algorithms for LUT networks rely heavily on SAT solving, limiting the number of optimization attempts and the size of the *substitution* subnetworks to one node [1]. Conversely, our method relies on circuit simulation to increase the number of substitution candidates and enables substitutions with more than one node. The experimental results show that the proposed method identifies optimization opportunities overlooked by other methods, improving 11 out of 23 best-known results in the EPFL synthesis competition and yielding a 3.46% area reduction compared to the state-of-the-art.**

***Index Terms*—FPGA, logic synthesis, area optimization, resubstitution, information graphs**

I. Introduction

AREA-oriented optimization is a crucial challenge in digital design. Indeed, efficient area utilization directly translates to cost savings in semiconductor manufacturing, correlates with more compact layouts, and improved wiring delays and power consumption. Despite decades of research, the ongoing demand for more efficient systems calls for more effort in area-oriented optimization strategies [2], [3].

This paper addresses the challenge of reducing the number of nodes in networks of *Look-Up Tables* (LUTs). This task is important for two reasons. First, Field-Programmable Gate Arrays (FPGAs) can be modelled as networks of LUTs, and minimizing the node count is essential for meeting resource constraints [4]. Second, in high-effort area-oriented design space exploration for standard cells, collapsing a network representation to an LUT network, restructuring it, and later deriving a new network in the original representation helps to escape local minima. Therefore, developing algorithms for optimizing and restructuring networks of LUTs broadly impacts the area-oriented optimization of combinational circuits.

Substitution, sometimes called *resubstitution*, is a method that tries to express (*resynthesize*) the function of a node using a set of candidate nodes already present in the network [5]. The transformation is accepted if the new implementation reduces the node count. The candidate nodes are named *divisors* because resynthesis generally occurs through a decomposition presenting the mathematical structure of a *division* [6]–[8].

Existing resubstitution-like algorithms for LUT networks, such as `mfs` [1], optimize circuit parts called *windows*. For each node of the network, called *pivot*, `mfs` solves a *dependency cut selection* problem, i.e., it identifies a subset of divisors that can be used to express the function of the *pivot*. The larger the window size, the more global information is used, resulting in higher optimization quality. However, `mfs` heavily relies on SAT solving for identifying *dependency cuts*, limiting the usable window size. Furthermore, the resynthesis subnetwork in `mfs` contains a single node.

Recently, *simulation-guided* approaches [6], [9], [10] have been proposed to harness non-local information beyond the constraints of conventional methods. These algorithms analyze functional simulations of the network to identify resubstitution candidates. Since exhaustive simulation of industrial-scale designs is prohibitive, only partial functional information can be analyzed, and obtained by simulating the network with a subset of its input patterns. SAT solving remains necessary to verify the correctness of the replacements. However, SAT-based equivalence checking is significantly more efficient than SAT-based *dependency cut* selection, enhancing scalability.

Simulation guided resubstitution algorithms have been proposed for simple circuit representations, like *And-Inverter Graphs* (AIGs). However, no such engine has yet been proposed for LUT networks because the higher expressiveness of their nodes' functionalities increases the complexity of efficiently addressing *dependency cuts* selection and resynthesis. This paper tackles this challenge, overcoming the limitations of current approaches, in particular `mfs`. This is achieved by devising novel techniques to solve two related problems:

1) How to perform *dependency cuts* selection without SAT?
2) How to enable resubstitution with more than one LUT?

Efficient solutions to the second problem are valuable because they facilitate the use of dependency cuts involving more divisors than the network's maximum fan-in size. For instance, given a network of 6-LUTs and a *dependency cut* made of 8 divisors, it might be possible to find a resubstitution using

A. Costamagna, A. Tempia Calvino, and G. De Micheli are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland (e-mail: andrea.costamagna@epfl.ch).

A. Mishchenko is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, US.

This research was supported in part by Synopsys Inc.

multiple 6-LUTs rather than a single one. An example can be observed in Figure 4.

The techniques proposed in this paper define a novel resubstitution engine with enhanced logic restructuring capabilities, which improve design space exploration. The experiments show that our heuristics identify optimization opportunities missed by other state-of-the-art engines, improving 11 of the best results in the EPFL competition. Furthermore, our heuristic proves effective for design space exploration, resulting in a 3.46% smaller area for the EPFL benchmarks when used as a replacement for `mfs` in an optimization flow.

The rest of this paper is organized as follows. Section II provides the preliminary background. Section III describes our engine. Sections IV and V present our algorithmic solutions to *dependency cuts* selection and LUT resynthesis. Section VI details the experiments, and Section VII concludes the paper.

II. Preliminaries

This section introduces the background and formalizes the theoretical notions underlying the proposed algorithms.

A. Boolean Functions And The Information They Contain

A *Boolean variable* x_i is a variable that takes values in the *Boolean space* $\mathbb{B} = \{0, 1\}$. The *literals* of a variable x_i are the functions $x_i, x_i' : \mathbb{B} \to \mathbb{B}$ that *distinguish* with their values when the variable is 1 from when it is 0. The input values for which a Boolean function is 1 are said to *satisfy* the function.

Let $\mathcal{X} = (x_1, \ldots, x_n)$ be an ordered set of Boolean variables. The n-dimensional Boolean space $\mathbb{B}^n = \{0, 1\}^n$ is the set of all the 2^n n-dimensional tuples obtained by assigning 0 and 1 to the variables in $\mathcal{X}$ in all possible ways. The *literals* x_i and x_i' partition the tuples in $\mathbb{B}^n$ into two sub-sets, based on whether x_i or x_i' is satisfied. Given k Boolean variables, a product of their literals is named *cube* C and is a Boolean function $C : \mathbb{B}^n \to \mathbb{B}$. A *cube* identifies the subset of tuples from $\mathbb{B}^n$ satisfying it. For instance, $\mathbb{B}^n$ is partitioned into 4 parts based on the satisfaction of the cubes $x_i'x_j'$, $x_i'x_j$, x_ix_j', and x_ix_j. A *minterm* of $\mathbb{B}^n$ is a cube of n variables, and it identifies a 1-dimensional subset of $\mathbb{B}^n$, i.e., a single point (or tuple).

A *completely specified* Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ associates each minterm M with a Boolean value $f_M \in \mathbb{B}$. The function partitions $\mathbb{B}^n$ into two sets: the *onset minterms* ($\mathcal{P}_1$) satisfy the function, and the *offset minterms* ($\mathcal{P}_0$) satisfy its negation[1]. The *information* of a Boolean function is the separation it induces in $\mathbb{B}^n$ by distinguishing $\mathcal{P}_1$ from $\mathcal{P}_0$. A completely specified function is in one-to-one correspondence with its onset minterms, so we use $\mathcal{P}_1$ and f interchangeably.

An *incompletely specified* Boolean function $f : \mathbb{B}^n \to \{0, 1, *\}$ is defined over a subset of $\mathbb{B}^n$, where the symbol $*$ denotes a *don't care* minterm, i.e., a minterm for which the function is undefined. A *partially specified* Boolean function $f : \mathbb{B}^n \to \{0, 1, *, ?\}$ is a function whose value is unknown for some minterms, named *don't knows* (?).

[1]Note that some authors use the term minterm for the onset minterms only.

B. Graphs

A *graph* $G = (V, E)$ is a pair (V, E) where V is a set of *vertices*, and E is a set of *edges*. A graph is *undirected* if the edges are unordered pairs, otherwise, it is *directed*. The *adjacency matrix* A of a graph $G = (V, E)$ is a Boolean matrix such that $A_{i,j} = 1$ if the edge (v_i, v_j) exists. The degree of a node is the number of edges containing the node. This paper considers two types of graphs: directed acyclic graphs to model combinatorial circuits and search spaces, and undirected colored graphs to represent and manipulate Boolean functions.

1) Directed Acyclic Graphs: A *path* $p = v_i \to \cdots \to v_j$ is a sequence of vertices connecting vertex v_i to vertex v_j, following the order induced by the edges set. A directed graph is *acyclic* if no path passes through the same node more than once. If there is an edge connecting node v_i to node v_j, v_i is in the *fanin* of v_j, and v_j is in the *fanout* of v_i. If there is a path from a node v_i to a node v_j, v_i is in the *transitive fanin* (TFI) of v_j, and v_j is in the *transitive fanout* (TFO) of v_i. The *maximum fanout free cone* (MFFC) of node x is the subset of nodes in the TFI of x such that every path from a node in the subset to the POs passes through x. When removing the node x, the MFFC can also be removed.

2) Undirected Colored Graphs: An undirected graph $G = (V, E)$ is *colored* if there is a label, named "color", associated with each vertex. An undirected graph $G = (V, E)$ is *bipartite* if its vertices can be divided into two disjoint and independent sets $\mathcal{P}_0$ and $\mathcal{P}_1$, that is, every edge connects a vertex in $\mathcal{P}_0$ to one in $\mathcal{P}_1$. The parts can be identified with a coloring in which no two adjacent vertices have the same color. An undirected bipartite graph is *complete* if each vertex of one part is connected to each vertex of the other part. The edges set of a complete bipartite graph is the cartesian product $E = \mathcal{P}_0 \times \mathcal{P}_1$.

C. Representing Combinatorial Circuits with Logic Networks

A *logic network* is a *directed acyclic graph* where the nodes are partitioned into three classes, named *primary inputs* (PIs), *primary outputs* (POs), and *internal nodes*:

- The PIs are nodes without *fanins* in the network.
- The POs are nodes without *fanouts* in the network.
- Internal nodes encode single-output Boolean functions.

A *k-input look-up table network* (k-LUT) is a logic network in which the nodes are k*-input look-up tables*.

A *structural cut* $\mathcal{C}$ of a node x in a logic network is a pair $\mathcal{C} = (x, \mathcal{L})$, where x is a node, called *root*, and $\mathcal{L}$ is a set of nodes, called *leaves*, such that 1) every path from any *primary input* (PI) to node x passes through at least one leaf and 2) for each leaf $v \in \mathcal{L}$, there is at least one path from a PI to x passing through v and not through any other leaf. A *reconvergence-driven cut* of size k is a structural cut of size k constructed to maximize the number of nodes and reconvergences included in the cut [7].

A *structural cut* $\mathcal{C} = (x, \mathcal{L})$ identifies a single-output subnetwork whose inputs are the leaves of the cut, and the output is the node x. This subnetwork represents a Boolean function that we name *cut funtionality*. If all the cut leaves are PIs, the cut functionality is the *global function* of the node.

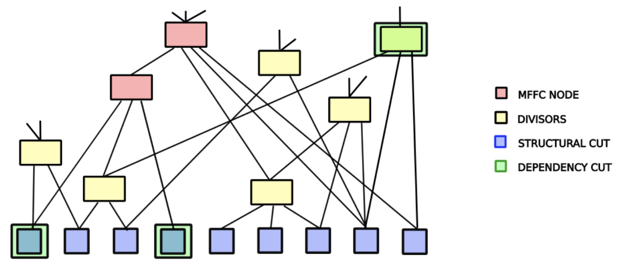


Fig. 1: A structural cut and a dependency cut.

A more general object is the *dependency cut* $\mathcal{C} = (x, \mathcal{L})$ that is not (necessarily) a structural cut in the current topology, but it has the potential to become one. Given a target node x, a *dependency cut* is a subset $\mathcal{C} = (x, \mathcal{L})$, where $\mathcal{L}$ contains nodes that are not in the TFO of the target and such that there is a function $f : \mathbb{B}^{|\mathcal{L}|} \rightarrow \{0, 1, *\}$ realizing $x = f(\mathcal{L})$. If such a function exists, it is called a dependency function, and the condition for its existence is simply the definition of a function, that in this context is the *dependency theorem* [10], [11]:

Theorem 1. *Let x be a node, and $\mathcal{L} = \{x_i\}_{i=1}^{K}$ be a set of nodes not in the TFO of x. The set $\mathcal{C} = (x, \mathcal{L})$ is a dependency cut iff the global functionalities of the nodes in $\mathcal{C}$ satisfy*

$$\forall M_i, M_j \in \mathbb{B}^n \qquad x_{M_i} \neq x_{M_j} \Rightarrow \exists x_l \in \mathcal{L} \text{ s.t. } x_{l,M_i} \neq x_{l,M_j}.$$

With an abuse of notation, we indicate the global function of a variable with the same name as the variable itself, so that x_{l,M_i} is the value of the Boolean function of node x_l evaluated at minterm M_i.

A structural cut satisfies Theorem 1, so it is a dependency cut. But a dependency cut is not necessarily a structural cut, as illustrated in Figure 1. Every dependency cut $\mathcal{C} = (x, \mathcal{L})$ (structural or not) is embedded in an environment, which is the surrounding network. Consequently, some minterms might never appear at the leaves of the cut. The *satisfiability don't care* set of the cut functionality f (DC_f) includes the tuples from $\mathbb{B}^{|\mathcal{L}|}$ never appearing at $\mathcal{L}$. In this case, f is *incompletely specified*.

D. Peephole Optimization

In logic synthesis, *peephole optimization* is an algorithmic technique that involves optimizing small sub-networks, named *windows*, to improve the overall circuit. Each window is built from a node, named *pivot*, and consists of the pivot's MFFC and a set of *candidate divisors*. This set is first initialized with the nodes on the paths between the MFFC leaves and the inputs of the window, which sets the boundaries for the network sub-portion considered for the optimization. Next, it is enlarged with nodes outside the TFI with both the fanins in the divisors' set.

Every peephole heuristic addresses three main problems:

1) Identify a *dependency cut* $\mathcal{C} = (x, \mathcal{L})$, for the *target* x.
2) Resynthesize the cut functionality
3) Replace the MFFC according to a cost function.

There are three types of peephole optimization, which differ from the information analyzed while attempting resynthesis:

1) Cut rewriting: This method relies on structural information by enumerating structural cuts for each pivot node.

2) Window-based resubstitution: this heuristic exhaustively simulates each window. The information analyzed is the function of the divisors with respect to the window's leaves.

3) Simulation-guided resubstitution: The information analyzed by this heuristic are approximations of the global functions of the divisors, named *simulation signatures*.

A *simulation signature* is a p-dimensional simulation vector of a node, obtained by simulating the network with p simulation patterns assigned at the network's inputs. The i-th entry of a simulation signature of a node is the value of the global function of the node on the i-th simulation pattern.

Simulation signatures enable analyzing global information in modern circuits, in which the dimensionality of $\mathbb{B}^n$ makes it unfeasible to exhaustively simulate circuits. Simulation signatures are approximations of the global functions of the nodes, which can be used during optimization.

E. Previous Work on Optimizing Look-Up Table Networks

Several existing algorithms address area optimization of LUT networks and are implemented in `ABC` [12].

Command `mfs` [1] is a window-based resubstitution algorithm using don't-cares to re-express the target node using a single k-LUT with reduced fanins number. The support selection problem is solved as an instance of a satisfiability problem, which checks if the dependency theorem is satisfied by a subset of k or fewer divisors.

Unlike `mfs`, our approach is not limited to a single k-LUT. Furthermore, our support selection method is based on *simulation signatures*, while SAT solving is only used to verify functional equivalence. In other words, our engine is simulation-guided [6].

Command `lutpack` [13] employs Boolean decomposition to re-express LUT sub-networks using fewer k-LUT. However, the support selection strategy corresponds to finding *structural cuts* and is agnostic of global *don't-cares*. Our technique considers global *don't-cares* and relies on the notion of *dependency cuts* rather than employing a *cut-rewriting approach*. Furthermore, our resynthesis algorithm enhances Boolean decomposition with a new strategy based on *don't-cares*.

F. Representing Boolean Functions with Information Graphs

In order to devise algorithms exploiting global functional information, we need effective ways of analyzing mutual dependencies between simulation signatures. *Information graphs* (IGs) represent the information of a Boolean function (Section II-A). First introduced by Józwiak [14], they offer a graphical representation of *sets of pairs of functions to be distinguished* (SPFDs), which are of great practical utility in resubstitution [15]–[18]. We propose a new formulation of IGs, at the basis of our algorithms.

Definition II.1. The *Information graph* (IG) of an incompletely specified Boolean function $x : \mathbb{B}^n \rightarrow \{0, 1, *\}$, is an undirected graph in which the vertices are the elements of

MINTERMS			FUNCTIONS		
M	x_1	x_0	x_1	x_0	x
0	0	0	0	0	0
1	0	1	0	1	1
2	1	0	1	0	1
3	1	1	1	1	$*$

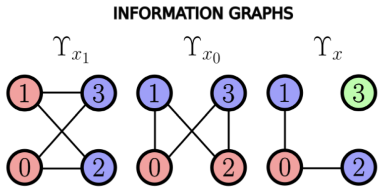


Fig. 2: Information graphs of simple two-input functions.

$\mathbb{B}^n$, and each *onset* minterm $\mathcal{P}_1^x$ is connected to each *offset* minterms $\mathcal{P}_0^x$, identifying a complete bipartite sub-graph:

$$\Upsilon_x = (V = \mathbb{B}^n, E = \mathcal{P}_0^x \times \mathcal{P}_1^x) \tag{1}$$

The edges represent the capability of the function to distinguish the *onset* minterms from the *offset* minterms. We define the following coloring for IGs:

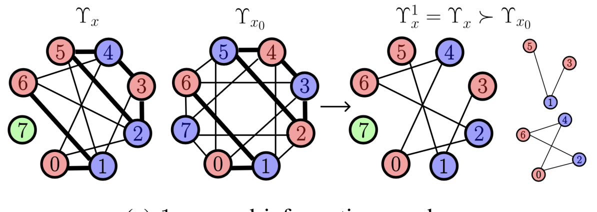
- Color 0: all the *offset* minterms, with non-zero degree.
- Color 1: all the *onset* minterms, with non-zero degree.
- Color 2: all *don't care* minterms, with degree zero.

Figure 2 illustrates the information graph of some simple two-input functions, represented as truth tables. x_1 and x_0 identify the encoding of the two independent variables of the Boolean space $\mathbb{B}^2$, and x identifies an incompletely specified Boolean function. For an *incompletely specified function*, the vertices with degree 0 are the *don't care* minterms, and the function is not required to distinguish them from other minterms. On the contrary, each *onset* (*offset*) minterm must be distinguished from the *offset* (*onset*) minterms. Hence, the degree of an *onset* (*offset*) vertex in the IG of an incompletely specified function is equal to the size of the offset (onset).

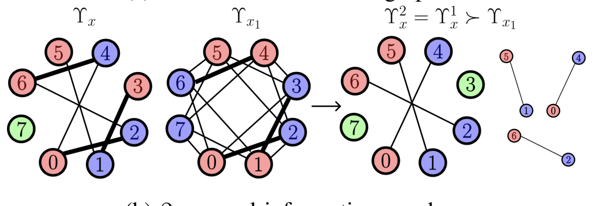
The algorithms in this paper rely on transformations of IGs that remove edges and swap onset with offset minterms. Hence, a vertex can undergo the following color changes:

$$\text{Color 2} \leftarrow \text{Color 0} \rightleftharpoons \text{Color 1} \rightarrow \text{Color 2}$$

G. Covering Information Graphs



(a) 1-covered information graph.



(b) 2-covered information graph.

Fig. 3: Covering information graphs.

Let us consider two Boolean functions $x, x_i : \mathbb{B}^n \rightarrow \{0, 1, *\}$, with IGs Υ_x and Υ_{x_i}. We consider the case in which the *don't care* set of x_i is contained in the *don't care* set of x, which is always true for the simulation signatures in our setting. The *edge covering* of Υ_x by Υ_{x_i} is the operation of removing from Υ_x the edges that are in common with Υ_{x_i}, i.e., removing from Υ_x all the minterm pairs that are distinguished by Υ_{x_i}. We name the result of this operation a 1-*covered* IG:

$$\Upsilon_x^1 \doteq \Upsilon_x \succ \Upsilon_{x_i} = (V = \mathbb{B}^n, E = E_0^1 \cup E_1^1) \tag{2}$$

Figure 3a represents the covering of an IG to obtain a 1-covered IG. The sub-graph identified by E_0^1 (E_1^1) contains the minterm pairs that x distinguishes, but x_i does not because they are in its *offset* (*onset*). In simple terms, this operation can be described as a *cofactoring* of the information graph with respect to the variable under consideration [5]. More explicitly:

$$E_0^1 = (\mathcal{P}_{0,0}^{x,1} \times \mathcal{P}_{0,1}^{x,1}) = ((\mathcal{P}_0^{x_i} \cap \mathcal{P}_0^x) \times (\mathcal{P}_0^{x_i} \cap \mathcal{P}_1^x)) \tag{3}$$
$$E_1^1 = (\mathcal{P}_{1,0}^{x,1} \times \mathcal{P}_{1,1}^{x,1}) = ((\mathcal{P}_1^{x_i} \cap \mathcal{P}_0^x) \times (\mathcal{P}_1^{x_i} \cap \mathcal{P}_1^x)) \tag{4}$$

where, the subset $\mathcal{P}_{q,b}^{x,t}$ is characterized by 4 attributes:

1) x is the node whose IG is being covered.
2) q is the index of the complete sub-graph identified by the partition, whose edges set is $E_q = (\mathcal{P}_{q,0}^{x,t} \times \mathcal{P}_{q,1}^{x,t})$.
3) b differentiates the onset minterms from the offset minterms in the complete sub-graph of the partition.
4) t identifies how many covering steps have been performed up to the definition of the subset.

This notation helps introduce the recursive expression of a t-covered IG, presented in the following. If all the sets $\mathcal{P}_{i,j}^{x,1}$ are non-empty, this operation results in two complete bipartite sub-graphs, in which we preserve the coloring of the graph that we are covering (Υ_x). Figure 3a explicitly shows these sub-graphs.

$$E_0^1 = (\mathcal{P}_{0,0}^{x,1} \times \mathcal{P}_{0,1}^{x,1}) \tag{5}$$
$$E_1^1 = (\mathcal{P}_{1,0}^{x,1} \times \mathcal{P}_{1,1}^{x,1}) \tag{6}$$

If $\mathcal{P}_{i,0}^{x,1}$ or $\mathcal{P}_{i,1}^{x,1}$ is empty, the minterms in the non-empty subset are colored as detached: there is no remaining information

$$E_i^1 = (\varnothing \times \mathcal{P}_{i,1}^{x,1}) \Rightarrow \mathcal{P}_{i,1}^{x,1} \mapsto \mathcal{P}_{i,1}^{x,1} \tag{7}$$
$$E_i^1 = (\mathcal{P}_{i,0}^{x,1} \times \varnothing) \Rightarrow \mathcal{P}_{i,0}^{x,1} \mapsto \mathcal{P}_{i,0}^{x,1} \tag{8}$$

The covering can be iterated using a set of IGs:

Definition II.2. Given a function $x : \mathbb{B}^n \rightarrow \{0, 1, *\}$ and an ordered set of functions $\mathcal{L} = (x_i)_{i=0}^{K-1}$, $x_i : \mathbb{B}^n \rightarrow \{0, 1, *\}$, the *covering process* $\Upsilon_x^0 \rightarrow \cdots \rightarrow \Upsilon_x^K$ is the sequence of transformations of Υ_x where each step t removes the edges that are present in Υ_{x_t}, generating a t-*covered* IG.

$$\begin{aligned} \Upsilon_x^0 &= \Upsilon_x = (V = \mathbb{B}^n, E = \mathcal{P}_{0,0}^{x,0} \times \mathcal{P}_{0,1}^{x,0}) \\ \Upsilon_x^t &= \Upsilon_x^{t-1} \succ \Upsilon_{x_t} \\ &= (V = \mathbb{B}^n, E = E_0^t \cup E_1^t \cup \cdots \cup E_{2^t-1}^t) \end{aligned}$$

where the edges are recursively defined as

$$E_{q=2m}^t = (\mathcal{P}_{q,0}^{x,t} \times \mathcal{P}_{q,1}^{x,t}) = (\mathcal{P}_0^{x_t} \cap \mathcal{P}_{m,0}^{x,t-1}) \times (\mathcal{P}_0^{x_t} \cap \mathcal{P}_{m,1}^{x,t-1})$$
$$E_{q=2m+1}^t = (\mathcal{P}_{q,0}^{x,t} \times \mathcal{P}_{q,1}^{x,t}) = (\mathcal{P}_1^{x_t} \cap \mathcal{P}_{m,0}^{x,t-1}) \times (\mathcal{P}_1^{x_t} \cap \mathcal{P}_{m,1}^{x,t-1})$$

Υ_x^t contains at most 2^t complete bipartite sub-graphs. For instance, Figure 3b shows a 2-covered information graph, having $3 \leq 2^2$ complete bipartite sub-graphs. Also in this case, if nodes become detached, their color is updated

$$E_k^t = (\varnothing \times \mathcal{P}_{k,1}^{x,t}) = (\varnothing \times \mathcal{P}_{k,1}^{x,t}) = \varnothing$$
$$E_k^t = (\mathcal{P}_{k,0}^{x,t} \times \varnothing) = (\mathcal{P}_{k,0}^{x,t} \times \varnothing) = \varnothing$$

Figure 3b shows that the second covering detaches one vertex.

If the edges set of a K-covered information graph Υ_x^K is empty ($||\Upsilon_x^K||_E = 0$), we say the IG is completely covered.

H. Dependency Cuts and Information Graphs Covering

Covering of information graphs is important because it offers an algorithmic technique for identifying *dependency cuts*, implied by the following corollary of Theorem 1:

Corollary 1. *Let x be a target node and $\mathcal{L} = \{x_i\}_{i=1}^K$ be a set of nodes not in the TFO of x. If the K-covered IG*

$$\Upsilon_x^K = \Upsilon_x \succ \Upsilon_{x_1} \succ \cdots \succ \Upsilon_{x_K}$$

is completely covered, then $\mathcal{C} = (x, \mathcal{L})$ is a dependency cut.

Proof. Let Υ_x and Υ_{x_i} be the IGs of the global functionalities of the target node x, and of any node $x_i \in \mathcal{L}$, and let us consider two minterms $M_i, M_j \in \mathbb{B}^n$. By definition II.1, if Υ_x has the edge (M_i, M_j), it must be true that $x_{M_i} \neq x_{M_j}$. In turns, Theorem 1 states that $\mathcal{C} = (x, \mathcal{L})$ is a *dependency cut* if and only if there is at least one node $x_l \in \mathcal{L}$ such that $x_{l,M_i} \neq x_{l,M_j}$. Hence, there is at least one IG Υ_{x_l} having the edge (M_i, M_j). Since the covering process removes all the common edges, and for each edge of Υ_x there is at least one node in $\mathcal{L}$ whose IG has the same edge, the K-covered information graph must be fully covered. □

Corollary 1 implies that the *dependency cut selection problem* can be addressed by identifying a set of nodes $\mathcal{L}$ whose IGs completely cover the IG of the target x.

I. Information Graph Representations

Practical simulation signatures typically range in size around $p \sim 2^{10}$. Consequently, representing IGs becomes challenging from a memory perspective, as it would require $\mathcal{O}(p^2)$ bits[2]. Recent advancements in resubstitution [19] have harnessed the insight that IGs can be constructively represented during a covering process. This led to the development of a specialized data structure tailored for performing covering processes on t-covered IGs, facilitating operations such as evaluating the remaining edges when covering Υ_x^t with the IG of a divisor $||\Upsilon_x^t \succ \Upsilon_{x_i}||_E$, and performing a covering step $\Upsilon_x^{t+1} = \Upsilon_x^t \succ \Upsilon_{x_i}$.

Building upon this foundation, the authors of the cited work proposed identifying *dependency cuts* through a covering process, where each step involves sampling a divisor from a probability distribution:

$$P(x_i; \beta, t) \propto e^{-\beta ||\Upsilon_x^t \succ \Upsilon_{x_i}||_E} \tag{9}$$

This distribution and its parameters were initially proposed as an ansatz for *dependency cuts* selection in simple representations, such as xor-and inverter graphs. In contrast, this paper introduces a novel approach: inferring models for the probability distribution from experimental data. This systematic data-driven method replaces predefined heuristics, enabling more efficient resubstitution in complex LUT networks and leading to improved optimization results.

[2]Assuming one bit per onset/offset minterm pair.

J. Markov Decision Process

A Markov Decision Process (MDP) is a discrete-time stochastic control process used for modelling decision-making in situations where outcomes are partly random and partly under the control of a decision maker [20].

The significance of MDPs emerges in several areas: notably, they are essential for structuring decision-making problems by clearly defining states, actions, rewards, and transitions. This structure makes it possible to handle stochastic environments where outcomes are uncertain [21].

The decision maker starts from an initial state and does an action. We consider the case where the relationship between the states and the action taken is a conditional probability, expressing the likelihood of an action given the current state. After a chosen action, the transition model maps the state-action pair into a new state. Here, we model the transition model as a deterministic map. Finally, for each action-state pair, the agent receives a reward. In short, an MDP presents the following features:

1) A set of possible states $\mathcal{S}$.
2) A set of possible actions $\mathcal{A}$.
3) A transition model $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$.
4) A real-valued reward function $R : \mathcal{S} \times \mathcal{A} \to [0, 1]$.
5) A policy $P(\cdot|s) : \mathcal{A} \to [0, 1]$.

Inspired by previous work on covering processes for resubstitution [19], in this work, we characterize an MDP for finding dependency cuts by adding one node at a time. This paper formalizes the covering process as an MDP and proposes a technique for inferring policies from data.

III. Simulation-Guided Resubstitution for LUT Networks

In this section, we discuss the structure of our algorithm for optimizing LUT networks. We adopt a simulation-guided approach, relying on the analysis of simulation signatures to identify resubstitution candidates, and on SAT solving to verify that functional equivalence is preserved. The engine relies on the solution of two sub-problems:

1) How to perform *dependency cuts* selection without SAT?
2) How to enable resubstitution with more than one LUT?

After explaining the role of these sub-problems in the algorithm, we discuss our algorithmic choices.

A. The Simulation-Guided Algorithmic Structure

Algorithm 1 illustrates our resubstitution engine. The algorithm starts by randomly sampling p tuples from the Boolean space $\mathbb{B}^n$, where n is the number of PIs of the circuit. Next, we simulate the input patterns, generating a simulation signature of size p for each node, and we perform peephole optimization one node at a time. Figure 4 illustrates the key algorithmic steps for each node. N is a parameter of the engine indicating the number of optimization attempts per node.

Algorithm 1: Resubstitution For k-LUT networks

```
Data: A $k$-LUT network $G$
Result: The $k$-LUT network after restructuring and
        optimization
1  $\mathcal{B}_{sig} \leftarrow$ Sample $p$ patterns at random from $\mathbb{B}^n$;
2  $\mathcal{B}_{cex} \leftarrow \varnothing$ ;
3  $\sigma_p \leftarrow$ Functional simulation of $G$ using $\mathcal{B}_{sig}$;
4  for $x \in G$ do
5      $\mathcal{C}_s = (x, \mathcal{L}_s) \leftarrow$Find a reconvergence driven cut;
6      Build a window from $C_s$;
7      for $N$ iterations do
8          $\mathcal{L} \leftarrow$ Solve the covering problem for $\Upsilon_x$;
9          $f(\cdot) \leftarrow$ Extract the functionality for $\mathcal{C} = (x, \mathcal{L})$;
10         $\mathcal{R} \leftarrow$ Resynthesize $f$;
11         if $Area(\mathcal{R}) \leq Area(MFFC)$ then
12             $x_{new} \leftarrow$Resynthesize $\mathcal{C}_d$;
13             if $x_{new}$ and $x$ are globally equivalent then
14                 Substitute $x$ with $x_{new}$;
15                 Go to the next node;
16             else
17                 Save the counter-example in $\mathcal{B}_{cex}$;
18                 if $|\mathcal{B}_{cex}|$ is equal to a word length then
19                     $\sigma_{64} \leftarrow$ Simulate $G$ using $\mathcal{B}_{cex}$;
20                     Replace oldest signatures with $\sigma_{64}$;
21                     $\mathcal{B}_{cex} \leftarrow \varnothing$ ;
22 return the optimized circuit;
```

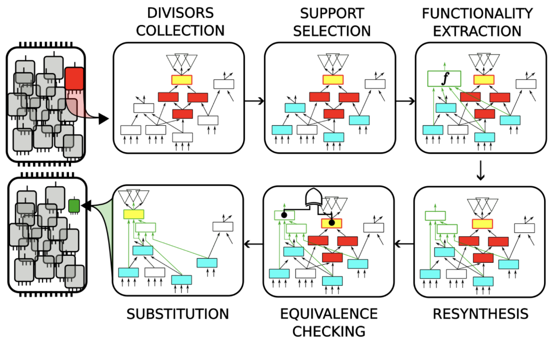


Fig. 4: Template of simulation-guided resubstitution for LUT networks, with a detail on LUT decomposition.

1) Window construction: For each node x, we build a *window* characterized by two parameters: W is the maximum size of the reconvergence-driven cut; D is the maximum number of divisors. As shown in Figure 4, during the window construction, we can compute the number of LUTs in the MFFC, which sets the potential gain of a resubstitution candidate.

2) Dependency-cut selection: Selecting a candidate *dependency cut* occurs through a stochastic exploration of the search space introduced in Section IV. The stochastic nature of the engine is such that different resubstitution attempts sample different supports.

3) Functionality extraction: Given a *dependency cut* $\mathcal{C}$ for a *target* node x, we extract $f : \mathbb{B}^{|\mathcal{L}|} \to \{0, 1, *\}$ by considering each minterm $M \in \mathbb{B}^{|\mathcal{L}|}$, and assigning it to onset, offset, and don't care set of the cut function based on its appearance and the value of the target signature. The result can be encoded in a $|\mathcal{L}|$-LUT, where $k \leq |\mathcal{L}| \leq K$, with:

- k: maximum fanin of the LUTs.
- K: maximum *dependency cut* size.

K is a parameter and k depends on the k-LUT network.

4) Resynthesis: We propose a decomposition-based resynthesis engine, which generates a k-LUT sub-network by decomposing the $|\mathcal{L}|$-LUT of the dependency function, while optionally taking the *don't cares* into account. Section V discusses the details of this algorithm.

5) Equivalence Checking and Substitution: If the area of the resynthesized sub-network can restructure the network without increasing area (or while improving area), we verify if the new node is functionally equivalent to the old one. In the case of a failure, the counter-example returned by the SAT solver can be used to update the simulation signatures, as suggested by the *simulation guided paradigm* [6], [9].

B. Support Selection, Resynthesis, and Generalization

Corollary 1 maps *dependency-cut selection* to set covering [22], [23]. The goal is to find *optimal dependency cuts*:

Definition III.1. A *dependency cut* found from the analysis of *simulation signatures* is *optimal* if it yields the smallest resynthesis sub-network generalizing to the *don't-knows*.

The condition of the generalization comes from the fact that a resubstitution candidate is not committed unless its output node is functionally equivalent to the target node for all possible input patterns, including the *don't knows*.

There is no guarantee that the minimum-size solution to the set covering problem using *simulation signatures* yields the optimal *dependency cut*. However, targeting the minimization of the *dependency cut* size is an educated guess. Indeed, on average, smaller supports yield smaller resynthesis sizes, with higher chances of meeting the size constraints imposed by the MFFC. According to the *Minimum Description Length principle* (MDL), smaller supports should also be chosen because they yield compact descriptions of the observations, which have a higher likelihood of generalizing to unseen patterns [24].

Previous works on partially specified Boolean functions showed that strong mutual dependencies between inputs and outputs can aid in a model's ability to generalize to unseen data [25], [26]. The coverage on an IG is a measure of such mutual dependency [14], which motivates investigating support selection algorithms where at each step we add variables to the support based on the analysis of IGs. Section IV addresses the challenge of devising algorithms to address this problem.

IV. Boolean Selection Of Dependency Cuts

Dependency-cut selection is the problem of identifying a set of nodes within the circuit that contains sufficient information

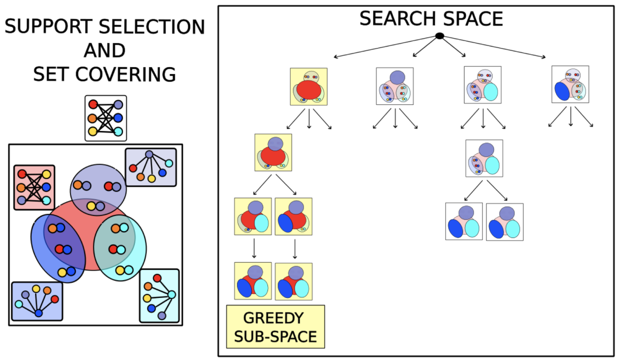


Fig. 5: Representation of the support selection problem as a set covering problem over the minterm pairs, and some sub-spaces of the search space.

to re-synthesize a target node. When limiting the solution size to K, the problem formulation is as follows:

K-DEPENDENCY CUT SELECTION

Given: 1) A target node x.
2) A set of candidate divisors $\mathcal{D} = \{x_i\}_{i=1}^{D}$.

Find a subset $\mathcal{L} \subseteq \mathcal{D}$, $|\mathcal{L}| \leq K$ satisfying Theorem 1.

Designing algorithms for efficiently addressing this problem is pivotal for building a successful resubstitution engine. The algorithms we propose in this section cope with the intractability of exhaustively exploring the solution space of this problem by confining the search to regions having a higher likelihood of containing a valid solution.

A. *Guiding Tree Search with Decision Processes*

We represent the search space for the *dependency cut selection problem* as a tree, as shown in Figure 5. The search tree has maximum depth K, which is the threshold value set for the support size. Each node in the tree, except for the root, corresponds to a divisor from the set $\mathcal{D} = \{x_i\}_{i=1}^{D}$, and the first level has one node per divisor branching from the root.

Each node at level l identifies a partial solution to the covering problem, consisting of collecting all the divisors on the path between the node and the root. Each node at level l branches to $D - l$ candidate divisors, which are the divisors from $\mathcal{D}$, not yet in the partial solution.

As a consequence of the branching, the total size of the tree search is $\binom{D}{K} = \mathcal{O}(D^K)$. For typical values, e.g., $D = 100$ and $K = 6$, it is impossible to enumerate through all the possible combinations in a window, especially since one such enumeration should be done for several windows of the order of the number of nodes in the initial network. Furthermore, in most windows, a solution of size K does not exist, so most nodes would experience the worst-case runtime $\mathcal{O}(D^K)$ without identifying optimizations.

For each window, the stringent runtime constraints imposed by logic synthesis force us to explore a fraction D^{-K} of the search tree. Consequently, devising efficient policies to guide tree search is of paramount importance. The goal is to devise an engine that exploits empirical evidence to guide the tree search. Since the model of the empirical observations might be limited, the engine should allow for some exploration capability in the limited number of runs available.

B. *Markov Decision Processes for Support Selection*

We describe the dependency cut selection problem as a *Markov Decision Process* (MDP) in which, at each step, the decision maker should choose which divisor to add to the dependency cut. The decision maker has N possible attempts and a maximum number of steps K for each attempt. At each attempt q, it starts from a state $s_q^0 = |\Upsilon_x^0; \Upsilon_{x_1}, \dots, \Upsilon_{x_D}\rangle$, characterized by the IGs of the target node and the divisors in the window. The selector transitions from one state to the next, and at the t-th iteration, the state is $s_q^t = |\Upsilon_x^t; \Upsilon_{x_1}, \dots, \Upsilon_{x_D}\rangle$, for some t-covered IG defined by the process $s_q^0 \rightarrow \cdots \rightarrow s_q^t$.

At step t of the q-th attempt, the decision maker takes an action $a_q^t \in \mathcal{A} = \mathcal{D}$, corresponding to choosing a divisor. The decision occurs by sampling an action using a conditional *policy*, that in this context is a conditional probability $P(\cdot|s_q^t) : \mathcal{A} \rightarrow [0, 1]$. This probability distribution should have a high value for the divisors that are likely to appear in a valid solution, and sampling allows us some degree of exploration to balance the limitation of the empirical model.

Let $a_q^t \sim P(\cdot|s_q^t)$ be the action sampled using the conditional policy. The transition model is a covering step, in which

$$T(s_q^{t+1}, s_q^t, a_q^t) = \delta(s_q^{t+1}, |\Upsilon_x^t \succ \Upsilon_{a_q^t}; \Upsilon_{x_1}, \dots, \Upsilon_{x_D}\rangle) \quad (10)$$

This decision process terminates after at most K steps. If termination occurs because the target information graph (IG) is fully covered, a *dependency cut* is identified. However, most possible subsets of size K do not yield a valid *dependency cut*, and it is unclear whether unsuccessful solutions give any insight into the usefulness of the divisors they contain. Consequently, the MDP for dependency cut selection has sparse rewards. Learning an optimal policy for each new sub-problem through *online learning* is significantly challenging in this context due to high *sampling complexity* [27], that is, a high number of random trials needed to learn an effective policy. Hence, online learning is impractical for the timing constraints of logic synthesis. Therefore, it is essential to develop *offline* learning strategies to create policies from known solutions to the dependency cut selection problem [28].

Let $\mathcal{L}_t = (x_1, \dots, x_t)$ be the partial solution, from which we want to choose the next divisor, and $s_q^t = |\Upsilon_x^t; \Upsilon_{x_1}, \dots, \Upsilon_{x_D}\rangle$ be the corresponding state. The goal is to identify a function $P(x_i \in \mathcal{D}|\mathcal{L}_t)$ expressing the likelihood that the divisor x_i should be added given the partial solution. As commonly done when modelling decision processes, for simplicity reasons, we assume that the process is *Markovian*, i.e., that we can identify the best next divisor just by considering the state of the problem at the current iteration. In this case, the function to identify becomes $P(x_i|\mathcal{L}_t) \simeq P(x_i|s_q^t)$.

C. *Analyzing the Policy of a Greedy Algorithm*

Greedy support selection (GSS) is the simplest *dependency cuts* selection algorithm that can be formalized as an MDP. At

TABLE I: ISCAS benchmarks: average percentage area variation after applying `mfs`, `mfs2`, and algorithm 1 with random, GSS, and enumeration support selection.

	mfs	mfs2	rnd	gss	enu
$\langle\delta_{100}\rangle[\%]$	-1.35	-1.46	-0.43	-1.94	-2.61

each step, the divisor whose IG covers most of the remaining edges is selected, and ties are broken at random:

$$P(x_i \in \mathcal{L}|s_q^t) \propto \delta(||\Upsilon_x^t \succ \Upsilon_{x_i}||_E - \min_{x_j \in \mathcal{D}} ||\Upsilon_x^t \succ \Upsilon_{x_j}||_E) \tag{11}$$

GSS is fast and simple but can fail to find some solutions. For instance, Figure 5 shows that GSS cannot find a solution when imposing a size constraint of 3 because it can only explore the sub-space of the search space highlighted in yellow. Consequently, it fails to find the existing solution with 3 divisors, for which non-greedy choices should be made. Consequently, we need policies with higher exploration capabilities.

To evaluate the possible improvements of using more refined policies, we compare GSS with an enumerative approach in a simple problem, and for small benchmarks. We consider the ISCAS benchmarks, after technology-independent optimization and 4-LUT mapping (`resyn2rs; fraig; st; dch; if -a -C 12 -K 4`). For each of them, we run Algorithm 1 setting the number of support sampling attempts to 100, for each pivot node.

As a baseline, we also report the result of choosing the next divisor at random at each branching point of the search tree. We limit the support size K to 4 to compare the performances with `mfs`. Table I reports the average results using two versions of `mfs`: `mfs` and `mfs2`. In both cases, we activate high-effort resubstitution, we set the number of windows to the number of nodes in the mapped network, and we allow 200 levels of depth increase for aggressive area optimization [1]. The key observations are the following:

1) GSS can beat the state-of-the-art in area optimization.
2) GSS misses optimization opportunities.

The result for enumeration gives an idea of what quality we can hope to achieve by refining the tree search exploration, but it does not scale to industrial designs. The challenge is to find more efficient ways to explore the search space, reducing the gap between enumeration results and an MDP-based heuristic.

D. Offline Learning of the Policy Function

Eq. 11 for greedy support selection assigns a non-zero cost only to the divisors that maximize the coverage of the IG. Inspired by the policy of GSS, we define a normalized cost:

Ansatz 1. *Let $\mathcal{D} = \{x_i\}_{i=1}^D$ be a set of divisors and $s_q^t = |\Upsilon_x^t; \Upsilon_{x_1}, \ldots, \Upsilon_{x_D}\rangle$ be a partial solution to an MDP. Then, the normalized cost*

$$\mathcal{H}(x_i, s_q^t) = \frac{||\Upsilon_x^t \succ \Upsilon_{x_i}||_E - \min_{x_j \in \mathcal{D}} ||\Upsilon_x^t \succ \Upsilon_{x_j}||_E}{||\Upsilon_x^t||_E - \min_{x_j \in \mathcal{D}} ||\Upsilon_x^t \succ \Upsilon_{x_j}||_E} \tag{12}$$

is a good metric to guide tree search exploration.

The normalized cost is a real number in the range $[0, 1]$. The divisors with 0 normalized cost cover most of the edges, i.e., the ones that GSS would choose. The divisors with 1 normalized cost are the ones not covering any edge, such as the ones selected in the previous steps.

We repeat the experiment in the previous section, but rather than committing the valid resubstitutions we save all the *dependency cuts* found by enumeration. Next, for each *cut* $\mathcal{C} = (\mathcal{L}, x)$, we artificially define a covering process. Starting from $s_q^0 = |\Upsilon_x^0; \Upsilon_{x_1}, \ldots, \Upsilon_{x_D}\rangle$, at each step, we take the divisors with the smallest normalized cost from $\mathcal{L}$, cover the graph, and transition to the next state. If the greedy approach can find the solution, the normalized cost of the divisor chosen at each step is 0. Otherwise, the normalized cost is some value higher than 0. By plotting the frequency of the normalized costs, we obtain the empirical frequency associated with the probability that a divisor has a normalized cost, given that it is chosen as the next support divisor: $\hat{P}(\mathcal{H}(x_i, s_q^t)|x_i \in \mathcal{L})$. Figure 6 shows that many valid supports can be identified by GSS. However, the normalized cost is larger than 0 in many cases, in correspondence with supports missed by GSS.

Figure 6 also shows the empirical distribution of the normalized costs $\hat{P}(\mathcal{H}(x_i, s_q^t))$. Using Bayes' Theorem, we estimate the posterior probability that a divisor should be included in the solution, given its normalized cost (Figure 6).

$$\hat{P}(x_i \in \mathcal{L}|\mathcal{H}(x_i, s_q^t)) \propto \frac{\hat{P}(\mathcal{H}(x_i, \Upsilon_x^t)|x_i \in \mathcal{L})}{\hat{P}(\mathcal{H}(x_i, s_q^t))} \tag{13}$$

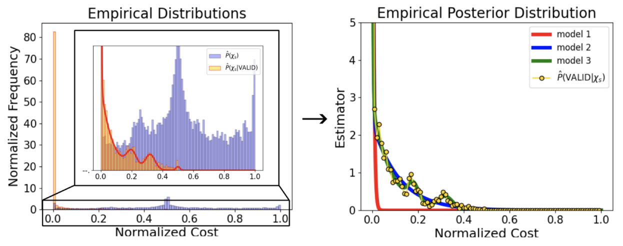


Fig. 6: Empirical distributions of the normalized cost, of the likelihood of a normalized cost given that the divisor belongs to a valid *dependency cut*, and the estimator of the probability that a divisor is valid, given its normalized cost.

In this setting, learning a policy involves defining a parametric model and using it to fit the empirical data on $\hat{P}(\cdot|\mathcal{H})$.

E. Parametric Models For The Policy

We fit the posterior distribution with three models. The first model mirrors the ansatz of the paper [19].

$$P_1(x_i \in \mathcal{L}|\mathcal{H}) = \alpha_1 e^{-\beta_1 \mathcal{H}}$$

As Figure 6 shows, this model fits well with the main behaviour but underestimates the probability that a *dependency cut* might distribute the information more evenly among the divisors. The second model is an *hyperexponential* [29]:

$$P_2(x_i \in \mathcal{L}|\mathcal{H}) = \sum_{i=1}^{2} \alpha_i e^{-\beta_i \mathcal{H}}$$

This model accounts for the fact that the distinguishing power of a valid support divisor either highly correlates with that of the target function or shares the information with other divisors. The third model encodes the distribution of this information with peaks at specific cost values.

$$P_3(x_i \in \mathcal{L}|\mathcal{H}) = \sum_{i=1}^{2} \alpha_i e^{-\beta_i \mathcal{H}} + \sum_{i=3}^{4} \alpha_i e^{-\frac{(\mathcal{H}-\mu_i)^2}{2\sigma_i^2}}$$

This model accounts for the fact that some values of the normalized cost are more likely than others (Figure 6).

Algorithm 2 shows how to use these probability distributions to guide the tree exploration for support selection. To avoid overfitting the parameters, we fit them on the ISCAS benchmarks and validate the models on the EPFL benchmarks. These models for the posterior distributions are not guaranteed

Algorithm 2: SUPPORT SELECTION<MODEL>$(x, \mathcal{D}; K)$

```
while trials < maxtrials AND !solution found do
    $t \leftarrow 0$ $\mathcal{L}_t \leftarrow \varnothing$;
    while $|\mathcal{C}| < K$ do
        evaluate the normalized costs $\mathcal{H}$;
        sample a divisor $d \sim P_{\text{MODEL}}(x_i \in \mathcal{L}|\mathcal{H})$;
        $\mathcal{L}_{t+1} \leftarrow \mathcal{L}_t \cup \{d\}$;
        $\Upsilon_x^{t+1} \leftarrow \Upsilon_x^t \succ \Upsilon_d$;
        $t \leftarrow t+1$;
        if $\mathcal{C} = (\mathcal{L}_t, x)$ satisfies Theorem 1 then
            return $\mathcal{C} = (\mathcal{L}_t, x) = (\mathcal{L}, x)$;
```

to extend to other circuit representations and optimization objectives. This paper only shows that fitting them with some benchmarks and for some target objectives generalizes to other benchmarks when optimizing them for the same target objectives. Instead, the fact that the normalized cost is a good optimization metric is a general observation.

Table II reports the results of the three methods for the EPFL benchmarks. These benchmarks were not used to fit the parameters of our algorithms, so they represent a valid test set. We observe a monotonic improvement when going from greedy to the second model. Instead, while the third model is the one that better fits the posterior probability, it does not generalize well to designs that were not available during training. In light of these results, which are consistent with the MDL [24], we prioritize the second model.

V. Look-Up Table Synthesis With Don't-Cares

Given a *dependency cut*, it is possible to obtain the dependency function $f : \mathbb{B}^K \to \{0, 1, *\}$ by looking at the simulation signatures and filling in the entries of a K-LUT based on the patterns appearing at the leaves of the *cut*. If a pattern does not appear, we treat it as a *don't-care*. This section discusses how to decompose this K-LUT into a network of k-LUTs, with $k < K$, while taking *don't-cares* into account. This is important because it enables resubstitutions exploiting dependency cuts of size $K > k$. If a dependency cut has a size larger than the maximum fan-in of the k-LUTs, we

TABLE II: Comparison of the state-of-the-art resubstitution with our engine, using the three policies for divisor selection. The results are shown for the EPFL benchmarks represented as 4-LUT networks. Each algorithm was run for 3 iterations.

design	mfs	mfs2	greedy	P_1	P_2	P_3
bar	1152	1152	1152	1152	1150	1152
div	4475	4475	4426	4420	4412	4399
log2	9690	9689	9618	9592	9580	9576
multiplier	7285	7285	7275	7273	7234	7244
sin	1828	1828	1806	1799	1790	1785
sqrt	7508	7508	7438	7383	7037	7003
square	5765	5765	5454	5409	5319	5294
cavlc	278	277	273	273	269	268
ctrl	51	50	47	48	44	46
i2c	439	433	439	435	435	437
int2float	78	78	84	84	84	84
mem_ctrl	15209	15552	15099	15184	15139	15145
priority	260	260	257	258	256	258
router	81	72	90	88	89	86
voter	2523	2525	2414	2371	2310	2322
	−1.56%	−2.16%	−1.99%	−2.21%	−3.22%	−3.18%

cannot perform resubstitution by adding a single node, because it would not be legal for the chosen technology. Instead, if we can express the cut functionality as a sub-network of k-LUTs, and this sub-network area is smaller than the current MFFC, the decomposition enables additional optimizations.

LUT SYNTHESIS WITH DON'T-CARES

Given: 1) A function $f : \mathbb{B}^K \to \{0, 1, *\}$.
2) A maximum fanin size is k

Find a kLUT network synthesizing f.

The support for *don't cares* information, enabled by using IGs, sets this method apart from previous resynthesis engines for LUTs, like the one used in Lutpack.

A. Information Graph Transformations

We start by defining some IG transformations needed to understand the proposed decomposition.

Definition V.1. Let Υ_x^t be a t-covered IG, and A be the adjacency matrix of the IG. The *adjacency preserving transformations* $\{\Delta_m\}_{m=0}^{2^t-1}$ are the transformations

$$\Delta_m(\Upsilon_x^t) = \{V = \mathbb{B}^n, E = \delta_m(E_0^t) \cup \cdots \cup \delta_m(E_{2^t-1}^t)\} \quad (14)$$

where three cases are possible

1) $\delta_m(E_i^t = \mathcal{P}_{i,0}^{x,t} \times \varnothing) = (\mathcal{P}_{i,0}^{x,t} \times \varnothing)$
2) $\delta_m(E_i^t = \varnothing \times \mathcal{P}_{i,1}^{x,t}) = (\varnothing \times \mathcal{P}_{i,1}^{x,t})$
3) $\delta_m(E_i^t = \mathcal{P}_{i,0}^{x,t} \times \mathcal{P}_{i,1}^{x,t}) = \begin{cases} (\mathcal{P}_{i,0}^{x,t} \times \mathcal{P}_{i,1}^{x,t}) & \text{if } m_i = 1 \\ (\mathcal{P}_{i,1}^{x,t} \times \mathcal{P}_{i,0}^{x,t}) & \text{if } m_i = 0 \end{cases}$

where m_i is the i-th bit of the binary representation of m.

These transformations are the IGs obtained by inverting the colors of the Color 0 and Color 1 vertices, according to the index of the transformation m. Since the transformation has the effect of swapping the colors in some of the bipartite sub-graphs, the adjacency structure, which is agnostic of the coloring, is preserved. Figure 7 shows 4 adjacency preserving transformations.

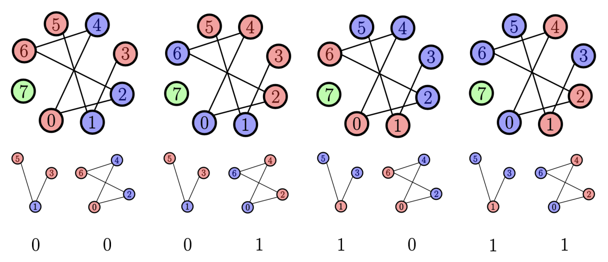


Fig. 7: Adjacency preserving transformations.

Definition V.2. Let Υ_x^t be a t-covered IG, and Δ_m an adjacency-preserving transformation. The *projection* operator Π is the operator collapsing the IG $\Delta_m(\Upsilon_x^t)$ into a Boolean function $\pi = \Pi(\Delta_m(\Upsilon_x^t))$. The Boolean function $\pi : \mathbb{B}^K \to \{0, 1, *\}$ is defined as

$$\pi_M = \begin{cases} 0 \text{ if vertex } M \text{ has } \textcolor{red}{\text{Color } 0} \\ 1 \text{ if vertex } M \text{ has } \textcolor{blue}{\text{Color } 1} \\ * \text{ if vertex } M \text{ has } \textcolor{green}{\text{Color } 2} \end{cases}$$

For instance, the projections for the example in Figure 7 are

x_2	x_1	x_0	$\Pi\Delta_0$	$\Pi\Delta_1$	$\Pi\Delta_2$	$\Pi\Delta_3$
0	0	0	0	1	0	1
0	0	1	1	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
1	0	0	1	0	1	0
1	0	1	0	0	1	1
1	1	0	0	1	0	1
1	1	1	*	*	*	*

(15)

Only half of the projected functions provide new information, since the other half is related to the first one by negation.

The result of the covering process is twofold:

1) From one covered IG, many functions can be derived, and their IGs have the same adjacency matrix, i.e., the same distinguishing power.
2) The covering process can enlarge the *don't-care set* of the original function $\mathtt{DC}_x \subseteq \mathtt{DC}_{\Pi(\Delta_m(\Upsilon_x^t))}$.

The set of minterms detached during the covering process corresponds to the vertices that undergo a coloring transformation into Color 2 and correspond to the set $\mathtt{DC}_{\Pi(\Delta_m(\Upsilon_x^t))} \setminus \mathtt{DC}_x$.

Definition V.3. Let $x : \mathbb{B}^K \to \{0, 1, *\}$ be a Boolean function, $\Upsilon_x^t = \Upsilon_{x_1} \succ \cdots \succ \Upsilon_{x_t}$ be a covering process, Υ_x^t the resulting t-covered IG, and Δ_m an adjacency-preserving transformation. These transformations are *support-reducing* if the support size after the projection is smaller than K.

The term *reduced support* is used for the support of the function obtained after a support-reducing transformation. For instance, Eq. 15 shows that the projection operation of the second adjacency preserving transformation is support-reducing since $\Pi(\Delta_1(\Upsilon_x^1))$ does not depend on x_1. Its reduced support is (x_2, x_1), and its reduced functionality now fits in a 2-LUT implementing the function $\Pi(\Delta_1(\Upsilon_x^1)) = x_2 \bar{\oplus} x_1$.

B. Decomposing a K-LUT into Two k-LUTs

The transformation discussed in the previous section results is an intuitive approach for detecting when a K-LUT can be decomposed into two k-LUTs, possibly sharing some fanins.

Theorem 2. *Let $x : \mathbb{B}^K \to \{0, 1, *\}$ be a Boolean function, $\mathcal{L} = \{x_i\}_{i=1}^n$ be a set of functions satisfying Theorem 1, and k a desired fanin size. If it is true that:*

1) *$K \le 2k - 1$.*
2) *There is a subset $\mathcal{L}_T \subset \mathcal{L}$, named top subset, $|\mathcal{L}_T| = k-1$ for which there is a support-reducing transformation.*
3) *The reduced support $\mathcal{L}_B \subset \mathcal{L}$ satisfies $|\mathcal{L}_B| \le k$.*

Then, x can be decomposed using two k-input functions.

$$x = g(\mathcal{L}_T, h(\mathcal{L}_B)) \qquad g, h : \mathbb{B}^k \to \mathbb{B} \tag{16}$$

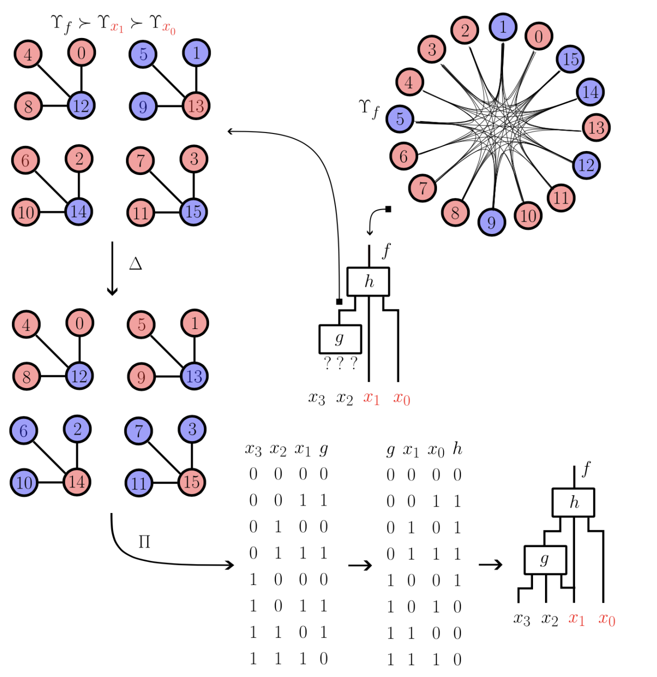
Proof. By definition of support-reducing transformation, the IG of the function $h(\mathcal{L}_B)$ covers the $(k-1)$-covered IG obtained with the covering process defined by the variables $\mathcal{L}_T$. Hence, the set $\mathcal{L}_T \cup \{h\}$ satisfies Theorem 1, implying the existence of a dependency function $g : \mathbb{B}^k \to \mathbb{B}$. $\square$

This remark provides an operational definition of the key engine of our decomposition, which is reported in Algorithm 3.

Algorithm 3: `2_decompose`(x)

```
$\chi \leftarrow$ sort_by_coverage$(x_1, \ldots, x_n)$;
for $\binom{n}{k-1}$ support subsets $\mathcal{L}_T = \{x_i\}_{i=1}^{k-1} \subset \mathcal{L}$ do
    $\Upsilon_x^{k-1} \leftarrow \Upsilon_x \succ \Upsilon_{x_1} \succ \Upsilon_{x_2} \succ \cdots \succ \Upsilon_{x_{k-1}}$;
    $m \leftarrow$ get_m_minimizing_ones();
    iter$\leftarrow 0$;
    while iter $\le 1 +$ effort $\cdot\, 2^{k-2}$ do
        $h \leftarrow \Pi(\Delta_m(\Upsilon_x^{K-1}))$;
        $\mathcal{L}_B \leftarrow$ get_support$(h)$;
        if $|\mathcal{L}_B| \le k$ then
            return $g(\mathcal{L}_T, h(\mathcal{L}_B))$;;
        iter$\leftarrow$mod(iter++,$2^{k-1}$)    $m$++;
```

The first step sorts the divisors by coverage of Υ_x, so that in the enumeration of all possible top subsets, we will first consider the input variables leading to the highest IG coverage. This choice was empirically verified to speed up synthesis. Next, we consider all the possible combinations of $k-1$ divisors as the input of the top LUT, and we cover the IG with them. Using theorem 2, a decomposition into two k-LUTs exists if and only if there is a k-input function covering the remaining edges of the IG. This function can be the projection of any adjacency-preserving transformation. Iterating through all the possible transformations would allow us to find a solution when present, but it requires an exponential number of trials in the worst case. To reduce the runtime effort, we introduce the parameter `effort`, in the range $[0, 1]$, where 0 corresponds to only one adjacency preserving transformation, and 1 considers all of them. To maximize the chances that a solution is found in the first few iterations, the first adjacency-preserving transformation we consider minimizes the number



x_3	x_2	x_1	g
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

g	x_1	x_0	h
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Fig. 8: Example of two-decomposition: for a candidate subsets of top variables, the covered IG is obtained, and a support-reducing transformation allows for the encapsulation of the reminder function into a 3-LUT, enabling decomposition.

of ones in the projected function. This choice detects many decompositions with the lowest effort (see Section VI-A).

Figure 8 shows an example. We represent the IG Υ_f of a 4-input function $f : \mathbb{B}^4 \to \mathbb{B}$ that we want to decompose into two 3-LUTs. The figure shows the case in which the attempted *top subset* is $\mathcal{L}_T = \{x_1, x_0\}$. After covering Υ_f with Υ_{x_1} and Υ_{x_0}, the 4 complete bipartite sub-graphs that can be covered by the function g are extracted. Next, the figure shows an adjacency-preserving transformation that is support-reducing after the projection step. The function depends on x_1, which is automatically detected to be a shared variable. Once obtained g, it is possible to extract the functionality of h that allows to recover the function f.

We attempt this decomposition every time the support size is $K \leq 2k-1$. When the decomposition fails, or when $K > 2k-1$, we try performing one step of top disjoint-support decomposition [30], and fallback to Shannon decomposition in case of failure. We use as the branching variable the one covering most edges in the IG, and we recursively apply the decomposition to the cofactors after updating their care sets with the branching variable. We discuss the details of these choices in the next sub-sections.

C. Decomposition with More Than Two k-LUTs

If the K-LUT cannot be decomposed into two k-LUTs, we can reduce the support with a Shannon decomposition:

$$f = x_i f_{x_i} + x_i' f_{x_i'} \doteq \texttt{ite}(x_i, f_{x_i}, f_{x_i'}) \tag{17}$$

where x_i is the *branching variable*, and f_{x_i} ($f_{x_i'}$) is the positive (negative) cofactor. The K-LUT will be decomposed into one 3-LUT for the multiplexer, and two $\tilde{K}$-LUTs, where $\tilde{K} \leq K-1$. Since we want to minimize the number of LUTs, it is beneficial to choose the branching variable resulting in reminder functions requiring the smallest number of LUTs to be synthesized. Also in this case, we use IGs:

$$x^* = \arg\min_{x_i} ||\Upsilon_f \succ \Upsilon_{x_i}||_E \tag{18}$$

The branching variable is the one whose IG covers most of the edges in the target IG. Since Eq. 18 relies on IGs, it can take don't cares into consideration during variable selection.

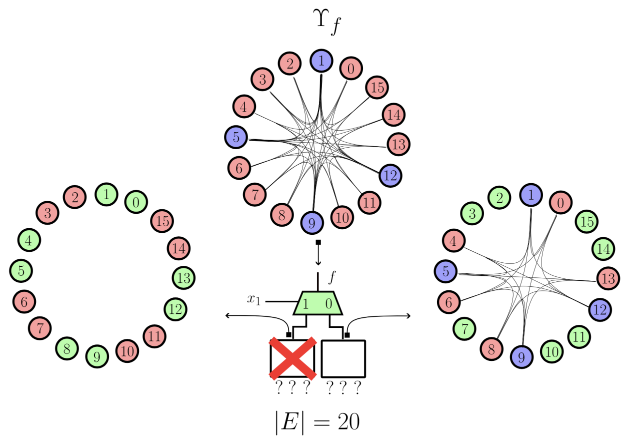


Fig. 9: Choosing a variable resulting in an unique reminder.

Figure 9 motivates this choice with an example. By choosing the variable x_1 over variable x_3, the number of remaining edges to cover are $||\Upsilon_x \succ \Upsilon_{x_1}||_E = 20 < ||\Upsilon_x \succ \Upsilon_{x_3}||_E = 24$. These edges are distributed between the two reminder functions. Since synthesis ends when an IG is completely covered, prioritizing branching variables that minimize the number of remaining edges is related to identifying higher simplicity of the reminder functions, which yields simpler decomposability. For instance, Figure 9 shows that the variable minimizing the number of remaining edges is the one requiring a single reminder, whereas the other candidate variable requires at least two more LUTs.

D. Collecting the Top-Down Decompositions

A Boolean functions $f : \mathbb{B}^K \to \{0,1,*\}$ is *top-down decomposable* on the variable x_i when

$$f = x_i \odot h(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_K) \tag{19}$$

with $\odot$ indicating a 2-input function. All the possible top-down decompositions can be obtained by listing the special cases of the Shannon decomposition: $f = x_i \lor f_0$, $f = x_i < f_0$, $f = x_i \leq f_1$, $f = x_i \land f_1$, and $f = x_i \oplus f_0$.

Since a Boolean function can be top-down decomposable in more than one variable, it is valuable to detect the cases in which the function can be top-down decomposed in up to $k-1$ variables. Indeed, let us consider the case in which $k = 3$ and f satisfies

$$f = x_i \odot_i (x_j \odot_j h(\mathcal{L} \backslash \{x_i, x_j\})) \tag{20}$$

Algorithm 4: Recursive Top-Down Decomposition

```
Data: A $K$-LUT for $f : \mathbb{B}^K \to \{0, 1, *\}$
Result: A top-down decomposed network of $k$-LUTs
1  $\mathcal{L}_T = \varnothing$, $\mathcal{S} = \{x_i\}_{i=0}^{K-1}$ and $h \leftarrow f$;
2  while $|\mathcal{L}_T| < k$, and $\mathcal{L}_T$ is updated do
3      for $x_i \in \mathcal{S}$ do
4          if $h_{x_i} = \bot$ or $h_{x_i} = \top$ then
5              $h \leftarrow h_{x_i'}$;
6          else if $h_{x_i'} = \bot$ or $h_{x_i'} = \top$ then
7              $h \leftarrow h_{x_i}$;
8          else if $h_{x_i'} = h_{x_i}$ then
9              $h \leftarrow h_{x_i'} + h'_{x_i}$;
10         if $h$ updated then
11             $\mathcal{L}_T \leftarrow \mathcal{L}_T \cup x_i$;
12             $\mathcal{S} \leftarrow \mathcal{S} \backslash x_i$;
13 $x_t \leftarrow$ decompose $h$;
14 $\mathcal{L}_T \leftarrow \mathcal{L}_T \cup \{x_t\}$;
15 $k$-LUT$\leftarrow$ extract top $k$-LUT from $\mathcal{L}_T$
```

Then, f can be cast in the form $f = g(\mathcal{L}_T, h(\mathcal{L}_\mathcal{B}))$. If $|\mathcal{L}_\mathcal{B}| \leq k$, this condition is a particular case of the decomposition into two LUTs. However, this technique becomes interesting when $|\mathcal{L}_\mathcal{B}| > k$, as it provides a way to reduce the support size with fewer LUTs compared to Shannon's decomposition.

Algorithm 4 illustrates the approach to identify a top-down decomposition step. In this case, the *don't care*-awareness comes from the fact that *don't cares* are used when performing the functional tests on the Boolean function.

VI. Experiments

This section discusses our experiments on optimizing LUT networks for area. After comparing our decomposition algorithm against existing methods, we investigate the effectiveness of our resubstitution algorithm.

A. LUT Decomposition

Table III shows the success rate of our decomposition when trying to decompose a K-LUT in a sub-network composed of two 4-LUTs. The method is indicated with IG44 to say that we use IG-based synthesis of a two nodes 4-LUT sub-network. The table considers *practical functions*, i.e., Boolean functions frequently appearing in modern hardware designs. Table III refers to the case in Algorithm 11 when the effort is set to the maximum value. Except for the 6-variables case, our algorithm identifies all the provably optimum results, and it does it faster than using a SAT formulation. The failure in the 6-variables case is because, for runtime reasons, our support reduction algorithm is not exact but heuristic, so it occasionally misses some optimization opportunities. Thanks to the small optimality gap, our heuristic achieves better results than *disjoint support decomposition* (DSD) and other methods used in `lutpack` (the first two lines of Table III).

To ensure scalability, we reduce the effort as much as possible. Table IV shows that the heuristics of Algorithm 11

TABLE III: Decomposition of the practical functions.

	5 vars (1233)		6 vars (7351)		7 vars (41071)	
	success	time[s]	success	time[s]	success	time[s]
DSD	55.31%	0.25	23.30%	1.81	16.52%	11.8
lpack [13]	91.08%	0.34	45.65%	2.11	18.70%	12.72
IG44	**96.67%**	**0.45**	**63.77%**	**31.64**	**20.86%**	**740.27**
SAT	96.67%	1.47	64.22%	34.98	20.86%	766.11

allow us to identify most opportunities for 2-decomposition in a single attempt. By increasing effort, other optimizations are identified. However, even in the lowest-effort configuration, the success rate is higher than in DSD and `lutpack`.

TABLE IV: Decomposition quality and effort.

effort	5 vars (1233)		6 vars (7351)		7 vars (41071)	
	success	time[s]	success	time[s]	success	time[s]
0%	**95.94%**	**0.033**	**60.94%**	**0.52**	**18.90%**	**10.26**
20%	96.59%	0.046	62.88%	1.64	19.32%	47.71
50%	96.67%	0.057	63.68%	3.48	20.26%	108.44

Finally, we test our algorithm in the presence of *don't-cares*. We take all the practical functions, for which there is no 2-decomposition, and we randomly generate a care-set. We compare our *don't-cares*-aware heuristic IG44$_*$ with IG44$_0$, which sets to 0 all the *don't-cares*, and IG44$_p$, which assigns a random value to them. Table V shows that leveraging *don't-cares* yields superior resynthesis quality.

TABLE V: Successrate of 2-decomposition with *don't-cares*.

	5 vars	6 vars (128350)	7 vars (260620)
IG44$_p$	28.39%	0.00%	0.00%
IG44$_0$	70.91%	4.94%	0.46%
IG44$_*$	**99.88%**	**92.60%**	**38.39%**

B. Resubstitution Statistics

In this experiment, we investigate the statistics of an optimization run. We consider the IWLS and EPFL benchmarks, initially represented as networks of *and-inverter graphs* (AIGs). We optimize them with one round of `resyn2rs`, and map them using the area-oriented LUT mapper with structural choices in ABC (`dch; if -a -K 4`).

Table VI reports the optimization statistics, including the number of attempted substitutions, the sizes of the resynthesis sub-networks, the number of successes, and the average number of *don't-cares* exploited by the accepted substitutions.

TABLE VI: Statistics for the EPFL and IWLS benchmarks.

		1-resub	2-resub	3-resub	4-resub	5-resub
GSS	valid	37606	2441	0	4	1
	trial	40697	3712	1	7	1
	$\langle\|\mathtt{DC}_x\|\rangle$	10%	28%	0%	32%	25%
P_1	valid	33419	2919	1	9	1
	trial	36447	4298	1	17	2
	$\langle\|\mathtt{DC}_x\|\rangle$	11%	28%	25%	27%	13%
P_2	valid	36281	3923	5	9	3
	trial	39306	5466	13	25	5
	$\langle\|\mathtt{DC}_x\|\rangle$	13%	30%	31%	26%	17%

TABLE VII: Design space exploration for 4-LUT-networks.

design	a_{ABC}	a_{NEW}	d_{ABC}	d_{NEW}	t_{ABC}	t_{NEW}
div	4339	4267	2148	2125	23.00	179.00
log2	9552	9472	153	212	364.00	640.00
mul	7156	7066	129	241	16.00	633.00
sin	1761	1673	78	120	29.00	504.00
sqrt	5244	4658	2009	2027	36.00	159.00
square	5232	4624	122	242	45.00	83.00
arbiter	4181	4020	31	31	20.00	122.00
cavlc	246	235	7	11	10.00	23.00
ctrl	45	42	4	10	3.00	4.00
i2c	379	338	8	10	7.00	29.00
int2float	67	66	7	7	2.00	4.00
memctrl	11962	10344	61	52	643.00	633.00
router	51	58	10	11	4.00	2.00
voter	2339	1950	18	30	49.00	325.00
	−11.15%	**−14.61%**				

C. High Effort Optimization of 4-LUT Networks

This experiment demonstrates the design-space exploration capabilities of our algorithm on the EPFL combinational circuits. As a baseline, we pre-process the benchmarks with ABC [12] to obtain small area k-LUT networks. First, we run the ABC script `resyn2rs`. Next, we apply SAT sweeping (`fraig`) to eliminate combinationally equivalent nodes. The networks are then mapped into 4-LUTs using an area-focused LUT mapper with structural choices (`dch; if -a -K 4`). We do not report the hypothenuse due to the high runtime needed to map this design using choices. The next section will discuss its optimization in the context of the EPFL competition. We compare two design-space exploration flows. The first flow iteratively runs `lutpack` and `satlut`. If no improvement is observed, we attempt logic restructuring with `mfs` and, subsequently, `mfs2`. If state-of-the-art resubstitution improves the area, the iterative `lutpack` and `satlut` process restarts. All algorithms are configured to focus solely on area optimization. A runtime limit of 10 minutes is imposed to terminate the flow if necessary. The second flow replaces `mfs` with our engine for single-node resubstitution and substitutes `mfs2` with our decomposition-based resubstitution. The higher runtime in Table VII is due to a longer optimization time before reaching convergence.

D. EPFL Best Results

The *EPFL combinational benchmark suite* comprises 23 combinational circuits used to benchmark logic optimization tools. The challenge is to derive 6-LUT network of the smallest size for each of these benchmarks. Notably, applying algorithms such as `lutpack`, `satlut`, `mfs`, or `mfs2` to these circuits does not yield any improvement. Further reducing the number of LUTs of these circuits proves the novelty of an optimization engine.

Table VIII shows the result of applying two optimization strategies to the best results, available in 2023. In both cases, we use the P_2-policy for support selection. In the first case, we run the resubstitution algorithm a single time, only considering dependency cuts of size 6. This single pass already identifies optimization opportunities. In the second case, we iteratively apply our heuristic with an initial maximum support size $K = 6$. If optimization fails, we increase K to 8 and attempt resynthesis into 6-LUTs. Upon success, we revert K to 6 and proceed. Optimization continues until convergence, also leveraging other state-of-the-art engines such as `mfs` [1], `lutpack` [13], and `satlut` [31]. Table VIII shows that this flow improved the best-known results for 11 out of 23 test cases from the competition. The large number of improved test cases confirms that our algorithm can identify optimization opportunities that cannot be found by previous methods.

TABLE VIII: Best area results for the EPFL benchmarks [32].

	OLD BEST		IG-RESUB$_{\times 1}$		FLOW$_{\infty}$	
Design	6-LUTs	Depth	6-LUTs	Depth	6-LUTs	Depth
div	3090	1100	3090	1100	3085	1102
hyp	36836	4384	36829	4547	36491	4633
log2	6076	243	6067	250	6012	257
mul	4330	178	4324	209	4314	208
sin	1053	86	1052	87	1023	110
sqrt	2983	1382	2980	1443	2966	1185
square	2959	170	2958	188	2935	200
i2c	177	9	177	9	176	9
memctrl	1708	14	1706	14	1694	14
priority	93	30	93	30	92	30
voter	1180	28	1179	30	1175	29

Of particular interest are the results for the `hypothenuse` and the `priority` test cases, the largest and the smallest benchmark in the suite, respectively. The optimization achieved for the `hypothenuse` proves the scalability of the method, while the optimization found for the `priority` shows that the transformations found by our engine are different to those found by previous approaches.

VII. Conclusion

This paper presents a resubstitution algorithm for combinational LUT networks, which has two novel features:

1) An ability to control the runtime of the divisor selection during resubstitution.
2) A decomposition strategy for synthesizing LUTs into networks of smaller LUTs.

The resubstitution engine, which integrates these heuristics, unlocks new optimization opportunities compared to state-of-the-art engines. Notably, our heuristics improves 11 of the best results in the EPFL competition. Since every network representation can be interpreted as a network of LUTs, the results of this paper can be extended to other representations in logic synthesis. Future works will discuss the benefits of the restructuring capabilities of our method to other network representations, including networks mapped into standard cells [33].

References

[1] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, “Scalable don't-care-based logic optimization and resynthesis,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–23, 2011.

[2] K. Tu, X. Tang, C. Yu, L. Josipović, and Z. Chu, “Logic synthesis,” in *FPGA EDA*. Springer, 2024, pp. 135–164.

[3] N. Grover and M. Soni, “Reduction of power consumption in fpgas-an overview,” *International Journal of Information Engineering and Electronic Business*, vol. 4, no. 5, p. 50, 2012.

[4] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in lut-based fpga technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2331–2340, 2006.

[5] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[6] S.-Y. Lee and G. De Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[7] A. M. R. Brayton and A. Mishchenko, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, vol. 6, 2006, pp. 15–22.

[8] R. K. Brayton, "The decomposition and factorization of boolean expressions," *ISCAS-82*, pp. 49–54, 1982.

[9] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2021.

[10] J. S. Zhang, S. Sinha, A. Mishchenko, R. K. Brayton, and M. Chrzanowska-Jeske, "Simulation and satisfiability in logic synthesis," *computing*, vol. 7, p. 14, 2005.

[11] J.-H. R. Jiang and R. K. Brayton, "Functional dependency for verification reduction," in *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*. Springer, 2004, pp. 268–280.

[12] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[13] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *International Conference of Computer-Aided Design (ICCAD)*. IEEE, 2008, pp. 38–44.

[14] L. Józwiak, "Information relationships and measures: an analysis apparatus for efficient information system synthesis," in *EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology (Cat. No. 97TB100167)*. IEEE, 1997, pp. 13–23.

[15] Y.-S. Yang, S. Sinha, A. Veneris, and R. K. Brayton, "Automating logic rectification by approximate spfds," in *2007 Asia and South Pacific Design Automation Conference*. IEEE, 2007, pp. 402–407.

[16] S. Sinha, *SPFDs: A new approach to flexibility in logic synthesis*. University of California, Berkeley, 2002.

[17] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for lut based fpgas and its applications," in *Proceedings of International Conference on Computer Aided Design*. IEEE, 1996, pp. 254–261.

[18] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

[19] A. Costamagna, A. Mishchenko, S. Chatterjee, and G. De Micheli, "An enhanced resubstitution algorithm for area-oriented logic optimization," 2024, accepted at the *International Symposium On Circuits And Systems (ISCAS)*.

[20] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[21] M. Van Otterlo and M. Wiering, "Reinforcement learning and markov decision processes," in *Reinforcement learning: State-of-the-art*. Springer, 2012, pp. 3–42.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[23] V. V. Vazirani, *Approximation algorithms*. Springer, vol. 1.

[24] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, 1978. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0005109878900055

[25] S. Rai et al, "Logic synthesis meets machine learning: Trading exactness for generalization," in *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021, pp. 1026–1031.

[26] A. Oliveira and A. Sangiovanni-Vincentelli, "Learning complex boolean functions: Algorithms and applications," *Advances in Neural Information Processing Systems*, vol. 6, 1993.

[27] J. Li and P. Gajane, "Curiosity-driven exploration in sparse-reward multi-agent reinforcement learning," *arXiv preprint arXiv:2302.10825*, 2023.

[28] D. Rengarajan, G. Vaidya, A. Sarvesh, D. Kalathil, and S. Shakkottai, "Reinforcement learning with sparse rewards using guidance from offline demonstration," *arXiv preprint arXiv:2202.04628*, 2022.

[29] A. Feldmann and W. Whitt, "Fitting mixtures of exponentials to long-tail distributions to analyze network performance models," *Performance evaluation*, vol. 31, no. 3-4, pp. 245–279, 1998.

[30] Bertacco and Damiani, "The disjunctive decomposition of logic functions," in *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. IEEE, 1997, pp. 78–82.

[31] B. Schmitt, A. Mishchenko, and R. Brayton, "SAT-based area recovery in structural technology mapping," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 586–591.

[32] L. Amarú, P. E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.

[33] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. De Micheli, "Improvements to boolean resynthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 755–760.

Andrea Costamagna received the B.Sc. degree in physical engineering from the Politecnico di Torino, Turin, Italy, in 2018, the joint M.Sc. degree in physics of complex systems from Politecnico di Torino (Italy) and Université Paris-Saclay (France) in 2020. Currently, he is working toward the Ph.D. degree at the Integrated Systems Laboratory at EPFL, Switzerland. His research interests include logic synthesis and optimization.

Alessandro Tempia Calvino received a B.S. degree in Computer Engineering from the Politecnico di Torino, Turin, Italy, in 2017, and an M.S. degree in Computer Engineering from the Politecnico di Torino, in 2020, and Télécom Paris, Paris, France, in 2021. He is currently pursuing a Ph.D. degree in Computer Science with the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland in the Integrated Systems Laboratory. His research interests include design automation, logic synthesis, and emerging technologies.

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis, formal verification, and machine learning.

Giovanni De Micheli is Professor and Director of the Integrated Systems Laboratory at EPFL Lausanne, Switzerland. He is a Fellow of ACM, AAAS and IEEE, a member of the Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies. He is member of the Scientific Advisory Board of IMEC and STMicroelectronics. Prof. De Micheli is the recipient of the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards.