Post-Mapping Resubstitution For Area-Oriented Optimization

Andrea Costamagna
EPFL
Lausanne, Switzerland

Alessandro Tempia Calvino
EPFL
Lausanne, Switzerland

Alan Mishchenko
UC Berkeley
Berkeley, California, USA

Giovanni De Micheli
EPFL
Lausanne, Switzerland

***Abstract*—The paper focuses on area minimization for circuits already mapped to a technology library, possibly under delay constraints. In contrast, traditional methods first optimize a technology-independent representation and then perform technology mapping to a library. A common assumption behind the traditional approach is that reducing the number of technology-independent nodes correlates with reduced area after mapping. This paper investigates the validity of this assumption and investigates the use of technology-dependent algorithms. We propose an area-oriented engine for mapped circuits that relies on a database of mapped sub-networks to achieve efficient resynthesis. Experimental results on EPFL and IWLS benchmarks after aggressive technology-independent area-oriented optimization and mapping show that the proposed method leads to an additional area reduction of 2.50% without worsening the delay.**

***Index Terms*—Optimization, Mapped circuits, Standard cells**

I. Introduction

DENNARD scaling of transistors enabled over fifty years of area-minimization of digital circuits, resulting in cheaper electronics, higher performance, and reduced power consumption [1]. While the demand for better performance continues to grow, financial and physical limits hinder delivering higher performance by using transistor scaling alone, and prolonging sustainable growth in computing power requires higher optimization effort at the design level.

State-of-the-art logic synthesis tools optimize technology-independent representations and map them to a technology library. In this paradigm, reducing the number of gates and logic levels in the technology-independent representation, named *subject graph*, is considered an effective heuristic for reducing area and delay after mapping. However, high-effort optimization of the technology-independent representation does not always correlate with improved mapped quality.

Technology-aware logic synthesis has recently gained substantial momentum as a new paradigm to improve the quality of mapped circuits [2]–[4]. This approach integrates technology-dependent information into logic optimization processes, thereby improving the quality of the resulting mapped circuits. By embracing this paradigm, this paper presents a novel technology-aware logic synthesis algorithm for area-oriented optimization, possibly under delay constraints.

This paper proposes an area-oriented optimization engine for circuits mapped using a library of *standard cells*. We adopt a *rewriting* approach, dividing the optimization process into two sub-problems: support selection and resynthesis. During support selection, the algorithm identifies divisors to serve as inputs for the resynthesis sub-network, while combining structural and Boolean techniques to maximize optimization opportunities. In the resynthesis phase, the sub-network is constructed. We achieve scalability by using a database of high-quality mapped sub-networks and enhance optimization quality by leveraging *don't care* information during resynthesis.

This research was supported in part by Synopsys inc., in part by SRC Contract 3173.001 "Standardizing Boolean transforms to improve quality and runtime of CAD tools".

Experiments on the EPFL and IWLS benchmarks confirm that aggressive area optimization of the technology-independent representation is a good heuristic, however, it does not always lead to the largest area reduction after mapping. This justifies the need for area optimization of mapped networks. We apply our technology-aware resubstitution algorithm to mapped designs after aggressive technology-independent optimization. Our method achieves an additional 2.50% average area reduction without impacting delay.

II. Background and Motivation

Combinational circuit synthesis is a fundamental problem when realizing efficient hardware for a computing system:

COMBINATIONAL CIRCUIT SYNTHESIS

Given:
1) A Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$.
2) A given library of gates L.
3) A cost function.

Find a network of gates from the library L to represent f while minimizing the cost function.

The cost function considered is the circuit area, with optional timing constraints. The end goal is a network, in which each node is a physical gate, placed at a location of a 2D grid, connected to other nodes through a layered wiring system, and characterized using an interconnect-dependent delay model. Designing this complex system while minimizing the cost function is done by the *design flow*, which progressively lowers the level of abstraction while optimizing increasingly more detailed circuit representations [5]. This section gives the background on area-oriented optimization at the *logic level* of abstraction.

A. Logic Circuit Representations

Logic synthesis is a key *design step* that transforms the specification of a function into a network of gates. The *front-end* of logic synthesis represents functional specifications as *technology-independent* logic models and optimizes them by algorithmic transformations. The *back-end* of logic synthesis uses the optimized representation obtained by the front-end to derive a *technology-dependent* representation, consisting of a network of gates from a technology library. The latter step

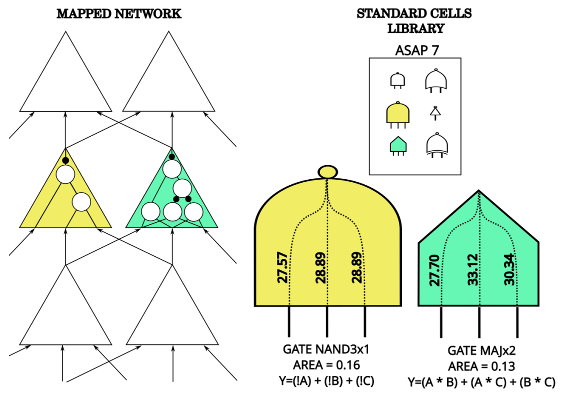


Fig. 1: Coverage of a subject graph with cells from a technology library. Each cell is characterized by its functionality, area information, and fixed pin-to-pin propagation delays.

is often simplified by assuming that the delay of the gates is *load-independent* under the gain-based delay model [6], i.e., each gate has a fixed worst-case pin-to-pin propagation delay.

A *Boolean network* is a directed acyclic graph, in which nodes represent logic gates and edges represent wires. A *technology independent representation* is a Boolean network, in which inverters have zero cost and nodes have simple functions. For instance, each node of an *and-inverter graph* (AIG) is a two-input and-gate. An *xor-and-inverter graphs* (XAIG) extends AIG with two-input xor-gates, while a *majority-inverter graph* (MIG) uses 3-input majority gates as primitives. These representations, particularly AIGs, are extensively used in logic synthesis to manipulate circuit specifications.

A *mapped network* is a Boolean network where each node is a cell from a *technology library*. The attributes of a node include its functionality, area, and pin-to-pin propagation delays. Figure 1 shows two cells from the $7nm$ technology library `asap7` [7]. *Technology-independent representations* can be seen as mapped networks where each node has a unit delay, unit area, and a simple logic function.

B. Strengths and Limitations of the Two-Step Optimization

State-of-the-art logic synthesis follows a two-step approach of optimizing an AIG, called the *subject graph*, followed by covering it with cells from a *technology library*. Figure 1 illustrates a portion of a *subject graph* covered with standard cells.

The motivation behind the two-step approach is the desire for better quality and scalability. Indeed, the simplicity of AIGs enables applying optimization algorithms to industrial size designs. In this way, the mapping algorithms can minimize the time spent on optimization and focus on the complexity of mapping to different technologies. This partition of the tasks also improves quality since the simplicity of AIGs compared to mapped networks allows for more intensive optimization within the same runtime, improving overall quality.

To obtain a mapped circuit optimized for a cost function, AIG optimization is guided by *technology-independent assumptions*, i.e., heuristic criteria estimating the correlation between AIG optimizations and improvements after mapping. A common assumption for area optimization is the following:

Assumption 1. *Reducing the number of nodes in the* subject graph *correlates with reduced area after mapping.*

This assumption has been fostering research on technology-independent optimization for decades. However, when striving to achieve better area, the following question arises:

Q1. *To what extent does Assumption 1 hold?*

Assumption 1 is related to the notion of *structural bias*: the structure of the mapped netlist strongly depends on the structure of the subject graph. While *structural bias* is essential for the success of the two-step process, mitigating it during mapping generally increases the quality of the resulting circuit [8]. Traditionally, this mitigation is achieved by increasing the capabilities of the mapper to evaluate alternative solutions in the mapped design space. A complementary approach comes from addressing the following question:

Q2. *Can we improve the circuit quality after mapping, with technology-aware area-oriented optimization?*

This challenge requires enriching technology-independent optimization algorithms with information about the technology.

C. Boolean Networks Terminology

In a Boolean network, if there is a path from a node x_i to a node x, then x_i is in the *transitive fanin* (TFI) of x, and x is in the *transitive fanout* (TFO) of x_i. The primary inputs (PIs) are nodes without fanins in the network and the primary outputs (POs) are nodes without fanouts in the network.

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. The MFFC of a node contains the portion of the circuit used exclusively to compute the functionality of x. When removing a node, its MFFC can also be removed.

A *structural cut* $\mathcal{C}$ of a Boolean network is a pair (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.

Given a cut $\mathcal{C} = (x, \mathcal{L})$, the paths connecting the leaves to the root identify a sub-network synthesizing a Boolean function named *cut functionality*. If all the leaves are PIs, the cut functionality is the *global function* of the node.

A *simulation signature* for a node x is a Boolean vector approximating the global function of node x. Analyzing the simulation signatures of a network enables identifying mutual relationships between the nodes' functions, which can be exploited to identify non-local optimization opportunities.

The *(controllability) don't-care set* for a cut $\mathcal{C}$ is the set of minterms in $\mathbb{B}^{|\mathcal{L}|}$ never appearing at the leaves of the cut. This gives the flexibility to choose an arbitrary value for the cut functionality at the missing input patterns so as to identify the function synthesized with the smallest-cost sub-network.

The presence of *don't cares* in a logic circuit is related to the existence of *reconvergences*, i.e. portions of the circuit in which the paths from a set of leaf nodes to a target node tend to diverge before reconverging to the target node. A *reconvergence-driven cut* of size k is a structural cut of size k constructed to maximize the number of nodes and reconvergencies included in the cut [9].

The *arrival time* of a node τ_x^A is the time at which the output signal of the node stabilizes after the inputs become stable. Given a delay constraint at the primary outputs, the *required time* τ_x^R of a node is the maximum allowed arrival time meeting the delay constraint. A node's *slack* is the timing flexibility of the node, defined as the difference between its *required time* and *arrival time*.

D. Characterizing Resubstitution

Boolean resubstitution is a transformation that attempts to *resynthesize* the function of a node using a set of candidate nodes, named *divisors*, already present in the network [9]. The transformation is accepted if the area of the new sub-network is smaller than the area of the MFFC.

Most resubstitution algorithms target one node at a time and identify a sub-network named *window*, consisting of the MFFC and a set of *candidate divisors*. This set is initialized with the nodes on the paths between the MFFC leaves and the leaves of a reconvergence-driven cut. Then, this set is enlarged until reaching a maximum size by adding any node outside the TFI of the target with both the fanins in the divisor set.

We characterize resubstitution as follows:

1) Window-based resubstitution: This implies that each window is exhaustively simulated, obtaining the *local function* of each divisor. Next, heuristic resynthesis tries to identify a resubstitution candidate, and if its area is smaller than the area of the MFFC, the substitution is performed.

2) Simulation-guided resubstitution: This means that the nodes' functions in the window are represented using simulation signatures, with the advantage of using global functional information. However, the signatures are approximations of the global functions, and equivalence checking is needed to verify the correctness of the transformation [10], [11].

After collecting functional information through simulation, a Boolean resubstitution engine uses it to check if the MFFC of the target node can be replaced with a smaller-area sub-network. State-of-the-art resubstitution engines use on-the-fly decomposition heuristics, simultaneously identifying useful divisors and adding them to the resynthesis sub-network. These resynthesis engines differ considerably from each other even for simple representations, e.g., AIGs, XAIGs, and MIGs. Consequently, on-the-fly decomposition is impractical for mapped networks since it requires developing separate resynthesis engines for each technology library.

E. Support Selection, Resynthesis, and Dependency Cuts

In contrast to on-the-fly decomposition, alternative approaches to resubstitution divide the process into two phases: divisor selection and synthesis [11], [12]. During the divisor selection phase, the goal is to identify a subset of nodes $\mathcal{C} = (x, \mathcal{L})$ that is not (necessarily) a structural cut in the current topology but has the potential to become one. Named *dependency cut*, this subset exists if there is a function $f : \mathbb{B}^{|\mathcal{L}|} \to \{0, 1, *\}$ such that $x = f(\mathcal{L})$. The synthesis phase then generates a sub-network implementing this function.

Dependency cuts can be found using *sets of pairs of functions to be distinguished* (SPFDs). Given a Boolean function $x : \mathbb{B}^n \mapsto \{0, 1, *\}$, its SPFD is a mathematical construct $\Upsilon_x : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$ encoding the Boolean function's ability to distinguish the points of its input space into *onset*, *offset*, and *don't care* sets [13], [14]. Theoretically, a set of nodes $\mathcal{L}$ identifies a *dependency cut* for a node x if the SPFDs of these nodes satisfy the covering condition [12], [13], [15]:

$$\Upsilon_x \subseteq \bigcup_{y \in \mathcal{L}} \Upsilon_y \Leftrightarrow \mathcal{C} = (x, \mathcal{L}) \text{ is a } \textit{dependency cut}. \quad (1)$$

A *structural cut* inherently qualifies as a *dependency cut*. Thus, enumerating *structural cuts* presents a straightforward means of identifying solutions to Eq. 1. However, *dependency cuts* are not always *structural cuts*. It's from identifying non-structural *dependency cuts* that non-local optimizations emerge.

Recent advancements in technology-independent resubstitution have led to the development of scalable algorithms focused on identifying dependency cuts through solving Eq. 1 for the SPFDs of simulation signatures. However, this approach lacks optimality guarantees regarding the size of the synthesized sub-network and faces scalability issues with large gate libraries, making it unsuitable for mapped networks.

Dependency cuts and the algorithms for finding them offer powerful tools for exploiting non-local dependencies. The functional nature of SPFDs allows their application to any representation, including mapped networks. This paper demonstrates that once a *dependency cut* is identified, its cut functionality can be extracted from simulation signatures. Heuristic resynthesis can then be replaced with database-based rewriting, offering a scalable approach with optimality guarantees. This method can be systematically applied to any new network representation, including *mapped networks*.

F. Boolean Matching and Rewriting

Let $f : \mathbb{B}^k \to \mathbb{B}$ be a completely specified Boolean function. The NPN-class of f is the set of Boolean functions that can be obtained from f by negating inputs and outputs and permuting inputs [16]. Similarly, the P-class of f is the set of Boolean functions that can be obtained from f by input permutation. The *representative* of a class is a function of the class.

Boolean matching [17] determines the NPN-class or P-class of a function, and efficient algorithms addressing this problem are known [18]. Boolean matching is used in *rewriting*, a key logic optimization technique based on extracting the cut functionality of a node and replacing the current sub-circuit with a pre-computed optimum version, stored in a database.

If a cut function has *don't cares*, it can match more than one class. *Boolean matching with don't cares* leverages this flexibility to match incompletely specified functions with the sub-circuits in a database having the minimum cost [17], [19].

G. Cut-Based Technology Mapping

The operation of modern technology mappers relies on two primary principles: Boolean matching and the utilization of *supergates*, which are mapped sub-networks constructed using gates from the technology library [20], [21]. The process of cut-based mapping can be summarized into five key steps:

1) Enumerate a set of *structural cuts* for each node [22].
2) Compute the Boolean function of each cut.
3) Match the functions with the *supergates*.
4) Sort the matches based on the target heuristic.
5) Select the optimal cover during a graph traversal.

It's important to note that the cuts considered during mapping are structural, meaning that introducing non-local rewiring in mapping is not feasible. While this limitation is partially addressed before mapping, further optimization opportunities may arise afterward. This paper aims to identify and exploit these opportunities.

Cut-based mappers mitigate *structural bias* by considering different implementations of local sub-structures in the subject graph, named *choices* [20], [23]. Using choices effectively identifies better-quality circuits because the mapper can explore a region of the mapped design space around the network obtained without choices. Optimization in the mapped design space is orthogonal to this approach: it further enlarges the explored region of the mapped design space while allowing for iterative refinements over multiple runs.

III. Technology-Aware Resubstitution

In this section, we introduce novel contributions enabling post-mapping design space exploration.The algorithms we developed rely on the observation that *dependency cuts* enable combining the global restructuring capabilities of *resubstitution* with database rewriting, which is scalable, and representation-independent. This observation results in different contributions. First, by integrating support for *dependency cuts*-selection, we enhance classical algorithms, thus broadening their applicability and effectiveness. Second, it leads to formulating a simple yet effective algorithm for optimizing the area of mapped networks. This algorithm addresses the following key sub-problems:

1) How to use *dependency cuts* for non-local rewriting?
2) How to synthesize mapped networks using *don't cares*?
3) How to efficiently account for timing information?

By proposing solutions to these sub-problems, our contributions transfer the expertise in technology-independent optimization to the mapped design space, thereby fostering advancements in post-mapping design space exploration.

A. Bridging Rewriting and Resubstitution

This section investigates the potential of enriching structural-based rewriting with *dependency cuts* selection. We consider the algorithm `rewrite` [19], which is implemented in the open-source logic synthesis library `Mockturtle` [24].

`Rewrite` explores circuits in a topological order, enumerates structural cuts for each node, and replaces the current node implementation when the cut functionality can be synthesized with a smaller area than the current implementation. This algorithm performs database-based rewriting, supports Boolean matching with *don't cares* (see Section II-F), and can be applied to AIGs, XAIGs, and MIGs. Furthermore, it can preserve the depth of the network when required.

To investigate the benefits of non-local restructuring in `rewrite`, we enrich this algorithm with *dependency cuts*-selection, resulting in a modified engine named `rewrub`. This engine integrates *window-based resubstitution* features in `rewrite`. For each target node, we construct a window and exhaustively simulate it. Next, we extract one dependency cut $\mathcal{C} = (x, \mathcal{L})$ using greedy-support selection. Analyzing the exhaustive simulation signatures of the windows, we obtain the incompletely specified cut functionality $f : \mathbb{B}^{|\mathcal{L}|} \rightarrow \{0, 1, *\}$, where the *don't cares* are present because some patterns do not appear at the leaves of the cut.

We consider three representations: AIGs, XAIGs, and MIGs. For each EPFL benchmark, we read the unoptimized AIG as the input representation, and optimize it with the two engines. When using `rewrite`, we optimize the graphs by enumerating nine 4-input structural cuts at each node and selecting the rewriting step that minimizes the number of nodes. We repeat the optimization with `rewrub`, which also enumerates eight 4-input *structural cuts* along with one *dependency cut*. The window is obtained with a *reconvergence driven cut* of size 8 and including at most 150 candidate divisors. We run both algorithms twice. The runtime comparison can be observed in table I, showing acceptable runtime increases when adding the dependency cut selection.

Table I shows the results: in all the considered setups, introducing *dependency cuts* increases the optimization quality. While `rewrub` always exploits *don't cares* by using *simulation signatures*, Boolean matching for the structural cut functionality can optionally use *don't cares*. Furthermore, we consider optimization with and without delay constraints. This motivating experiment shows that *dependency cuts* enable combining the global restructuring capabilities of *resubstitution* with database rewriting. Using don't cares does not help `rewrub` to the degree it helps `rewrite`. This is because `rewrub` can identify most of the available *don't care*-based optimizations.

While global restructuring using technology-independent representations can be addressed by running resubstitution after rewriting, this approach is highly representation-specific and not applicable to mapped networks (Section II-E). In contrast, our formulation enables resynthesis through database-based rewriting, which makes the approach scalable and representation-independent. Furthermore, since modern mappers only rely on structural cuts, this experiment motivates exploring optimization after mapping.

B. Optimization Engine

To enable novel optimization opportunities within mapped circuits, we rely on resubstitution to leverage non-local functional information, not-accessible during mapping. The novelty of this engine lies in integrating *dependency cuts* with database-based rewriting, which allows us to perform resubstitution on networks mapped onto standard cells. Additionally, we incorporate structural cuts into graph traversal to account for structural optimization opportunities that can emerge after non-local restructurings. Algorithm 1 outlines the proposed engine, which depends on the following parameters:

- τ^R_{max} is the maximum required time at the outputs.
- P is the allocated size for each simulation signature.
- C_w is the maximum input size of the window.
- D_w is the maximum size of the window.
- E is the number of enumerated structural cuts.
- N is the maximum number of resubstitution attempts.

If no delay constraint is required, we impose $\tau^R_{max} = \infty$.

The algorithm starts by generating a set of P-dimensional simulation signatures for the network's nodes, and extracting the timing information if delay constraints are imposed. Next, for each node we construct a window, and attempt resynthesizing the target node to reduce the area occupied by its MFFC. Three types of resubstitution are attempted, as detailed in the Section III-C. A resubstitution candidate is successful when its reward is larger or equal than a threshold value ϵ. Setting $\epsilon = 0$ enables logic restructuring even when the substitution does not result in area reduction, which can be beneficial for design space exploration [25]. When resubstitution is successful, we update the simulation and timing information.

This algorithm can be applied to any network representation for which a database of high-quality sub-networks can be constructed. Hence, in addition to enabling resubstitution for networks mapped to standard cells, this algorithm offers a new resubstitution approach that is representation-independent, a desirable feature in logic synthesis development [26].

TABLE I: Rewriting with *don't cares* based on *structural cuts* and *dependency cuts*. A indicates the area, d indicates the depth, δ indicates the percentage improvement compared to the value pre-optimization, and t indicates the average run time.

preserve depth	*don't cares*	library	$\delta A_{rewrite}[\%]$	$\delta A_{rewrub}[\%]$	$\delta d_{rewrite}[\%]$	$\delta d_{rewrub}[\%]$	$t_{rewrite}[s]$	$t_{rewrub}[s]$
no	with	AIG	-11.98	$\mathbf{-14.22}$	-4.90	-4.04	0.65	0.80
		XAIG	-21.03	$\mathbf{-22.76}$	-8.30	-7.60	0.58	0.73
		MIG	-20.02	$\mathbf{-22.63}$	-15.90	-12.71	0.74	0.91
	without	AIG	-9.77	$\mathbf{-14.16}$	-1.08	-4.14	0.44	0.74
		XAIG	-18.55	$\mathbf{-22.56}$	-3.97	-7.40	0.36	0.62
		MIG	-16.85	$\mathbf{-22.00}$	-10.96	-15.10	0.60	0.83
yes	with	AIG	-11.21	$\mathbf{-13.58}$	-6.48	-6.60	0.62	0.77
		XAIG	-19.88	$\mathbf{-21.74}$	-10.00	-10.70	0.54	0.68
		MIG	-18.09	$\mathbf{-20.59}$	-14.38	-13.08	0.83	2.07
	without	AIG	-8.82	$\mathbf{-13.43}$	-2.48	-6.56	0.26	0.43
		XAIG	-17.22	$\mathbf{-21.60}$	-5.57	-10.57	0.21	0.36
		MIG	-14.87	$\mathbf{-19.92}$	-9.82	-15.38	0.43	0.68

Algorithm 1: Technology-Aware Resubstitution

```
Data: A mapped circuit with required time $\tau^R_{max}$
Result: A new mapped circuit optimized for area
$\mathcal{B}_{sig} \leftarrow$ Sample $P$ patterns at random from $\mathbb{B}^n$;
$\sigma_P \leftarrow$ Functional simulation of $G$ using $\mathcal{B}_P$;
for $x \in G$ do
    if $\tau^R_{max} \neq \infty$ and $x$ is marked then
        update the required times $\tau^R$;
    Build a window of input size $C_w$ for node $x$;
    $\mathcal{C}_s, \mathcal{R}_s \leftarrow$ Find the best structural cut;
    $\mathcal{C}_d, \mathcal{R}_d \leftarrow$ Dependency cut from local information;
    if ($\mathcal{R}_s \geq \epsilon$ or $\mathcal{R}_d \geq \epsilon$) then
        $x_{new} \leftarrow$ Resynthesize the best cut;
        Substitute $x$ with $x_{new}$
    else
        for $N$ iterations do
            $\mathcal{C}_d, \mathcal{R}_d \leftarrow$ Find a candidate dependency cut;
            if satisfying reward ( $\mathcal{R}_d \geq \epsilon$) then
                $x_{new} \leftarrow$Resynthesize $\mathcal{C}_d$;
                if $x_{new}$ and $x$ are globally equivalent
                 then
                    Substitute $x$ with $x_{new}$;
                    Go to the next node;
                else
                    Save the counter-example in $\mathcal{B}_{cex}$;
                    if $|\mathcal{B}_{cex}|$ is equal to a word length
                     then
                        $\sigma_{64} \leftarrow$ Simulate $G$ using $\mathcal{B}_{cex}$;
                        Replace the oldest signatures;
                        $\mathcal{B}_{cex} \leftarrow \varnothing$ ;
    if successful resubstitution and $\tau^R_{max} < \infty$ then
        Update the arrival times $\tau^A$;
        Mark the TFI of $x_{new}$;
Return the optimized circuit;
```

C. Combining Different Cut-Selection Strategies for Efficiency

We employ three cut-selection techniques to capitalize on the strengths of different approaches.

First, we enumerate E structural cuts. For each cut, we assess the area of the MFFC portion contained between the cut's leaves and the target node. We determine the *reward* as the difference between this area and the area of the circuit synthesizing the cut functionality, obtained through *Boolean matching* with pre-computed sub-networks in a database.

The second cut-selection technique focuses on identifying *dependency cuts* through the analysis of the exhaustive simulation signatures of a window. After local simulation of the window's divisors, we rely on their SPFDs to find a *dependency cut*. Since optimizations involving MFFC nodes are already accounted for in the structural exploration, we restrict candidate divisors to nodes outside the MFFC. *Boolean matching with don't cares* identifies the smallest sub-network in the database that is compatible with the cut functionality, and we compute the *reward* as the difference between the area of MFFC and that of this sub-network.

The third cut-selection technique also targets identifying *dependency cuts* from the analysis of simulation signatures, but it relies on global simulation signatures. Consequently, it can identify optimization opportunities missed by the other methods by leveraging global *don't cares*. This step is attempted N times. Each attempt requires six steps, as shown in Figure 2:

1) A *dependency cut* $\mathcal{C} = (x, \mathcal{L})$ is selected.
2) The cut functionality $f : \mathbb{B}^{|\mathcal{L}|} \rightarrow \{0, 1, *\}$ is extracted.
3) The cut functionality is resynthesized.
4) We verify if the new implementation reduces the area.
5) We verify if functional equivalence is preserved.
6) We substitute the node with the new implementation.

The equivalence checking step is necessary because this approach relies on simulation signatures approximating the global node functionality. Therefore, unlike the other cut-selection methods, the substitution candidate is not guaranteed to preserve the functional equivalence of the circuit. Upon failure, the SAT solver provides a counter-example, that is a minterm from $\mathbb{B}^n$ disproving the validity of an optimization found with the current approximations of the network nodes' functionalities. To enhance scalability, we allocate fixed memory for signatures, simulating the network when the number of counter-examples reaches the length of a machine word (64 bits) and overwriting the oldest word in the signature.

Given the significantly faster performance of the first two strategies compared to the third one, if any of the cuts identified in these stages results in successful rewriting, we commit the best candidate and skip the simulation-guided step.

D. Functionality Extraction

Let us consider a window for which we have simulation signatures, which can represent the complete truth table or its approximation.We identify a *dependency cut* by solving the set covering problem defined in Eq. 1. We solve set covering with

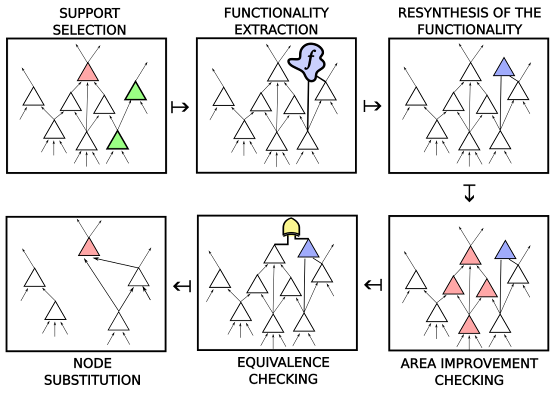


Fig. 2: Core of the resubstitution engine.

TABLE II: On-the-fly mapping and database matching.

design	$\Delta A_{map}[\%]$	$\Delta A_{match}[\%]$	$t_{map}[s]$	$t_{match}[s]$
c432	-2.69	-2.24	7.57	0.28
c1908	-1.95	-3.60	39.70	0.30
c3540	-1.66	-1.75	71.29	0.34
c5315	-0.62	-0.48	138.16	0.45
c6288	-0.15	-0.25	244.93	0.38
c7552	-0.36	-0.76	135.53	0.36

a greedy algorithm [12], [27]. The algorithm is randomized, so it can generate different *dependency cuts* if called multiple times. It is worth mentioning that randomization is intrinsic in the greedy set covering algorithm, which is the underlying algorithm used to solve support selection [27]. Since greedy set covering algorithm is an approximation algorithm with optimality guarantees, the presence of randomization should not be seen as a limitation, but as a feature of the engine. In any case, the algorithm is seeded, so that the stochasticity is only pseudo-random.

Given a candidate *dependency cut* $\mathcal{C} = (x, \mathcal{L})$, we extract its function from the simulation signature of its leaves $\sigma_{\mathcal{L}}$. Functionality extraction requires three steps:

1) Check which minterms of $\mathbb{B}^{|\mathcal{L}|}$ are present in $\sigma_{\mathcal{L}}$.
2) Save the corresponding value of σ_x in a truth table.
3) Consider the missing patterns as *don't cares*.

This process returns the cut functionality $f : \mathbb{B}^{|\mathcal{L}|} \to \{0, 1, *\}$, ready for synthesis. If the signatures come from exhaustive window simulation, the *don't cares* are guaranteed to be correct. Otherwise, they may stem from the suboptimality of simulation signatures.

TECHNOLOGY-AWARE RESYNTHESIS

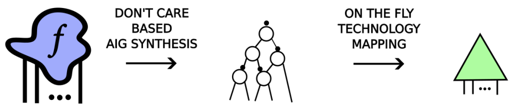
Given: 1) A function $f : \mathbb{B}^n \to \{0, 1, *\}$.
2) A library of standard cells.

Find a circuit with a small area representing f.

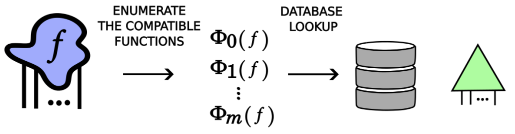
E. Synthesis of Mapped Networks with Don't Cares

Leveraging *don't cares* in the synthesis phase is important to increase the efficiency of resubstitution. Figure 3 illustrates the two heuristics discussed in this section.

The first possibility to achieve *don't cares*-aware synthesis is to adopt the two-step approach: we fist perform *don't care*-aware synthesis of an AIG, and we then map it on-the-fly. At each step, if the support size is smaller than 5, Boolean matching with *don't cares* completes AIG synthesis. Otherwise, a decomposition step combining disjoint support decomposition and shannon decomposition is performed to reduce the support size [28]. We choose the branching variable as the one with with the highest correlation with the function being decomposed, as this metric maximizes the number of don't cares in the reminder function. Finally, on-the-fly technology mapping generates the dependency sub-network.



(a) AIG synthesis followed by on-the-fly mapping.



(b) Boolean matching and database look-up.

Fig. 3: Synthesis strategies exploiting *don't cares*.

The alternative approach is to rely on a database of mapped networks for *completely-specified* functions and use *don't cares* during *Boolean matching*. Since inverters have a cost, the database contains one network for each *P-class* of the 4-input Boolean functions. We find the optimum-size AIG of each *P-representative* and map it with area-oriented technology mapping using the library.

For Boolean matching with *don't cares* on mapped sub-networks, simple enumeration proves effective and reasonably fast. For instance, statistics collected when optimizing the EPFL benchmarks show that the average number of *don't cares* per optimization is upper-bounded by 2, corresponding to an average number of database look-ups of 4 per optimization attempt. When presented with an incompletely-specified Boolean function f, we systematically enumerate all possible completely specified functions by allocating the *don't care* minterms to either the onset or the offset, and we commit a sub-network if its area is smaller than the current MFFC.

The comparison in Table II shows the significance of database rewriting and minimizing on-the-fly mapping: the overhead incurred by allocating the mapper results in reduced mapping effort, adversely affecting both quality and runtime.

To maintain delay awareness during area optimization, we cannot focus solely on area or delay. We therefore populate the database with "good-enough" solutions derived from the two-step logic synthesis paradigm (Section II-B) on a smaller scale, without guaranteeing optimality. Two factors contribute to this lack of optimality: 1) the database excludes gates with more than 4 inputs, and 2) we employ the two-step approach. This choice is justified by two observations. First, gates with many inputs have larger areas, making them less likely to be successful resubstitution candidates. Second, the 4-input database contains almost 4000 substructures, and experimental results show that optimizations are possible without fully exploiting the features of the technology library.

The goal is not to rewrite with the optimal sub-network but to use "good-enough" substructures. This choice is acceptable since local optimum rewriting does not guarantee the best overall optimization, especially during multiple optimization runs. Additionally, storing a database for gates with more than 4 inputs is impractical. If gates with more than 4 inputs are necessary, AIG synthesis with on-the-fly mapping can be used.

F. Timing Analysis and Minimizing Updates

Given a delay constraint on the required time τ^R_{max}, delay tracing is required to guide the optimization and verify if a candidate transformation satisfies the delay constraints.

The first step of the delay tracing is to collect the nodes of the network in a topological order. Next, the nodes are visited in this order and the arrival time of each node is computed as $\tau^A_n = \max_{x_i \in FI(n)} \tau^A_{x_i} + d_{x_i}$. If τ^A_{CP} is the arrival time of the critical path, we initialize the required times of the POs to the value $\tau^A_{CP} + \Delta t$. Next, we traverse the graph in a reverse topological order while annotating each node with the required time $\tau^R_n = \min_{x_i \in FO(n)} \tau^R_{x_i} - d_{x_i}$.

Effective resubstitution influences the arrival times of the TFO of the target node and the required times of the TFI of the new sub-network. While we must update the arrival times after every optimization step, the required times should be updated when necessary to minimize the impact of the timing updates on the algorithm's runtime. We observe that the required time of a node is needed only if it is a resubstitution target at a later stage. Hence, we mark the TFI nodes of each successful resubstitution, and if any of these nodes is the target node at a later stage, we update the required times of the entire network and remove the marks from the nodes.

IV. Experiments

This section presents experiments with technology-aware optimization using the $7nm$ standard cell library `asap7` [7]. For technology mapping, we utilize the state-of-the-art mapper `emap`, implemented in `Mockturtle` [24]. The results are collected on a Linux machine with an Intel `i7-1365U` CPU.

A. Validity of Technology-Independent Assumptions

In this experiment, we investigate the correlation between node minimization in the *subject graph* and reduced area after technology mapping. We consider an aggressive optimization flow, which runs the following area-oriented commands and scripts in `ABC`: `rw`, `rs`, `rf`, `resyn2rs`, and `compress2rs`. Before applying any optimization, the AIG is functionally reduced using command `fraig`. As soon as one heuristic successfully reduces the number of AIG nodes, we map the AIG to technology and plot the number of AIG nodes and the area. We use one-pass commands `rw`, `rs`, `rf` at the beginning to ensure slower convergence to a minima, to better visualize the correlation between technology-independent optimization and area after mapping. We iterate this procedure as long as the proposed heuristics lead to changes in the subject graph.

Figure 4 shows the typical trends observed on the EPFL and IWLS benchmarks. For most benchmarks, at least in the first few optimization steps, we observe a good correlation between AIG minimization and area reduction. For instance, in highly non-optimized AIGs, such as `mem_ctrl` (top-left of Figure 4), the correlation between AIG minimization and area reduction is consistently strong, without significant fluctuations. However, there are benchmarks in which the

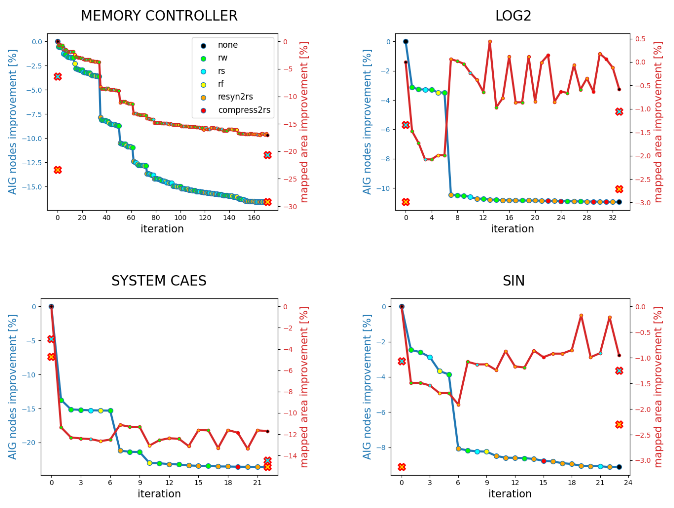


Fig. 4: Correlation between the node count in the subject graph and the area after mapping. The plots reflect typical optimization trends observed for the EPFL and IWLS benchmarks. The crosses represent the results of post-mapping optimization.

initial correlation between the optimization metrics becomes noisy after a few iterations and additional improvements in the AIG size can have an adverse effect when area after mapping increases by several percent. This is true for `log2`, `system_caes`, `sin` in Figure 4, and other test cases from the EPFL and IWLS benchmark suites.

These observations agree with the existing literature on array-based designs [29], but observing similar behaviors for standard cells-based design is more surprising due to the presence in the technology library of small gates with small area, offering higher flexibility to follow the updates of the subject graph compared to the unitary cost of the nodes in array based designs. Figure 4 also suggests that `resyn2rs` is the source of the noticeable "jumps" in the behavior.

B. Post-Mapping Area Optimization

To evaluate the proposed technology-aware resubstitution algorithm, we consider two points in the optimization process described in Figure 4:

1) Start: when the subject graph is not optimized.
2) End: when the AIG could not be further optimized.

We run the proposed resubstitution in two ways:

1) Single traversal of the graph.
2) Iterative traversal of the graph until convergence.

The percentage improvement, compared to the initial mapped network, is shown in Figure 4 by crosses. In the case of `mem_ctrl`, `log2`, `system_caes`, and `sin`, despite the high-effort AIG optimization, the proposed algorithm can achieve further area reduction. Furthermore, it is interesting to observe that in `mem_ctrl`, `log2`, and `sin`, the best optimization result is not achieved with the two-step process, but rather by directly optimizing the network mapped using the unoptimized subject graph.

C. Area under Delay Constraints

This experiment proposes a detailed analysis on the EPFL and IWLS benchmarks. Also in this case, we consider the

TABLE III: The worst-case analysis of the improvement of technology-aware resubstitution. The benchmarks are those with less than 200K nodes from the EPFL and IWLS benchmark suites. The benchmarks are optimized with high-effort AIG optimization before mapping. The best possible delay is used as the delay constraint.

Design	$A_i[nm^2]$	$\delta A_i^1[\%]$	$\delta A_i^\infty[\%]$	$A_e[nm^2]$	$\delta A_e^1[\%]$	$\delta A_e^\infty[\%]$	$D_e[ps]$	$\delta D_e^1[\%]$	$\delta D_e^\infty[\%]$	$t_e^1[s]$	$t_e^\infty[s]$
div	3914.60	−13.83	−22.19	1296.90	−5.05	−9.53	60248.23	0.00	0.01	4.82	60.04
sqrt	1372.25	−11.84	−15.70	1171.15	−3.04	−5.24	78957.63	−0.04	−1.22	4.01	63.66
arbiter	557.84	−14.36	−39.17	557.84	−14.08	−43.85	999.95	0.00	−23.82	1.78	17.53
cavlc	36.17	−1.19	−2.90	34.27	−0.15	−1.20	247.45	0.00	0.00	0.20	1.52
mem_ctrl	2547.32	−7.63	−17.56	2063.01	−5.56	−12.25	1649.46	−5.79	−10.64	19.26	184.25
aes_core	1198.55	−3.59	−5.50	1106.60	−0.65	−1.47	434.52	−2.07	−0.02	6.28	118.31
des_perf	5182.91	−3.61	−4.47	4615.59	−0.44	−1.08	584.17	0.00	0.00	73.93	215.58
ethernet	4411.85	−2.83	−3.62	3123.35	−0.39	−2.41	588.34	0.00	0.00	51.24	216.18
iwls05_i2c	66.32	−6.72	−9.92	49.96	−1.28	−1.26	288.07	0.00	0.00	0.20	0.39
sasc	40.46	−1.58	−2.37	31.72	−0.54	−1.39	191.00	0.00	0.00	0.15	0.61
simple_spi	54.77	−4.51	−6.46	41.58	−0.58	−1.88	287.00	0.00	0.00	0.17	1.48
spi	205.45	−2.25	−7.81	167.59	−0.88	−1.69	489.49	0.00	0.00	0.38	4.02
systemcaes	611.55	−2.52	−5.50	530.89	−3.08	−3.60	784.00	0.00	0.00	1.06	10.42
systemcdes	172.17	−5.81	−11.76	142.22	−0.79	−1.58	530.29	0.00	0.00	0.32	3.70
usb_phy	28.82	−5.97	−11.69	23.97	−0.83	−1.96	179.06	−7.79	−7.11	0.15	0.74
		−5.07%	**−9.68%**		**−1.06%**	**−2.50%**		**−0.62%**	**−1.27%**	8.58	38.67

subject graphs when they are unoptimized and when they are highly optimized. High-effort optimization is achieved by iteratively applying `resyn2rs` and `compress2rs` until the *subject graph* cannot be reduced, and merging equivalent nodes with the command `fraig` before applying any script. After the high-effort optimization, we map the network using area-oriented mapping and optimize the network with the most stringent delay constraints, i.e., we do not allow for any delay increase on the critical path. Table III shows the following quantities for this experiment:

- A_i: The mapping area using an unoptimized AIG.
- A_e: The mapping area using an optimized AIG.
- D_e: The mapping delay using an optimized AIG.
- $\delta Q_{i,e}^1$: Improvement after one optimization round.
- $\delta Q_{i,e}^\infty$: Improvement after iterative optimization.

We use fixed *simulation signatures* of size 2^{10}, and the windows are limited to at most 10 inputs and 256 divisors.

Table III reports the results for the EPFL and IWLS benchmarks. We consider 39 designs whose *subject graphs* initially have less than 200K nodes due to the high runtime of the flow including both AIG optimization and mapping. The average values at the bottom of the tables are computed for all 39 benchmarks, but we only report the most remarkable results for space limitation reasons. Table III shows that post-mapping resubstitution leads to noticeable reductions in area while reducing or preserving the worst-case delay.

One specific benchmark to discuss is the `arbiter`. At the *subject graph* level, neither `resyn2rs` nor `compress2rs` can optimize it. On the other hand, our experiment shows that performing optimization on the mapped design space unlocks impressive optimization opportunities. Statistics on this benchmark show that the mapped network exhibits a large number of *don't cares*, which increase the number successful matches during resubstitution.

D. Design Space Exploration

In the last experiment, we investigate design space exploration for mapped circuits. Table IV shows the results for the EPFL benchmarks. The average improvements are over all the EPFL benchmarks, excluding the *hypothenuse*. We optimize the subject graphs with two iterations of the script "`fraig; compress2rs; resyn2rs`". Next, we technology map the subject and optimize it under stringent delay constraints. In one case (column δA^∞), we iterate optimization until convergence. In the other case (column $\delta A^{5\times5}$), after 5 iterations of technology-aware resubstitution, we unmap the network resulting in an AIG, remove redundancies, and balance it [30]. This step attempts moving to a different region of the design space. Finally, we map back to technology and restart area-oriented optimization. We report the best encountered results where each benchmark also satisfies the delay constraints. This experiment shows that our engine effectively introduces logic restructuring in mapped networks, with direct benefits in design space exploration.

TABLE IV: Mapped design space exploration.

Design	$A[nm^2]$	$\delta A^\infty[\%]$	$\delta A^{5\times5}[\%]$	$t^\infty[s]$	$t_e^{5\times5}[s]$
bar	149.13	−0.04	−3.24	0.50	7.88
div	1302.07	−9.96	−15.54	91.97	75.89
sin	289.47	−0.24	−1.41	2.58	35.02
sqrt	1171.15	−3.99	−5.98	28.14	89.16
arbiter	557.84	−45.59	−55.49	11.74	19.71
cavlc	34.53	−0.96	−1.13	0.85	6.90
ctrl	5.90	−2.71	−3.90	0.29	4.92
i2c	59.23	−0.95	−1.08	0.62	6.09
mem_ctrl	2164.98	−13.96	−11.38	187.81	256.89
priority	27.66	−0.29	−2.75	0.30	5.34
		−4.23%	**−5.47%**		

V. Conclusion

This work proposes a *technology-aware* resubstitution algorithm for post-mapping area-oriented optimization. The approach is motivated by the diminishing correlation between technology-independent and technology-dependent optimization after a few iterations. We customize the proposed resubstitution method to work for *standard cell* designs. Experiments on the EPFL and IWLS benchmarks show that applying our method after aggressive logic minimization and area-oriented technology mapping further reduces area by 2.5% on average and up to -43.85% for some test cases. The results are obtained with the delay constraints. These encouraging results suggest a new line of research: applying Boolean methods in technology-aware logic optimization.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] L. Amarú, P. Vuillod, J. Luo, and J. Olson, "Logic optimization and synthesis: Trends and directions in industry," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1303–1305.

[3] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 352–359.

[4] A. T. Calvino, A. Mishchenko, H. Schmit, E. Mahintorabi, G. D. Micheli, and X. Xu, "Improving standard-cell design flow using factored form optimization," in *Proc. DAC*, 2023.

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

[6] B. Hu, Y. Watanabe, and M. Marek-Sadowska, "Gain-based technology mapping for discrete-size cell libraries," in *Proc. DAC*, 2003.

[7] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, 1997. [Online]. Available: https://doi.org/10.1145/264995.264996

[17] F. Mailhot and G. De Micheli, "Technology mapping using boolean matching and don't care sets." in *EURO-DAC*, vol. 90, 1990, pp. 212–216.

[18] X. Zhou, L. Wang, and A. Mishchenko, "Fast exact npn classification by co-designing canonical form and its computation algorithm," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1293–1307, 2020.

[19] A. Tempia Calvino and G. De Micheli, "Scalable logic rewriting using don't cares," in *Proc. DATE*, 2024.

[20] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, 2005, pp. 519–526.

[21] B. Robert, M. Alan, C. Satrajit, X. Kam, and T. Wang, "Technology mapping with boolean matching, supergates and choices."

[22] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.

[23] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *Trans. CAD*, vol. 16, no. 8, pp. 813–834, 1997.

[24] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage *et al.*, "The epfl logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.

[25] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[26] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarù, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[27] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[28] 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.

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

[30] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 375–382.