IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 25, 2020, accepted November 30, 2020, date of publication December 15, 2020,
date of current version December 31, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3045014

Extending Boolean Methods for Scalable Logic Synthesis

ELEONORA TESTA[1,2], (Member, IEEE), LUCA AMARÚ[2], (Member, IEEE), MATHIAS SOEKEN[1], (Member, IEEE), ALAN MISHCHENKO[3], (Senior Member, IEEE), PATRICK VUILLOD[4], PIERRE-EMMANUEL GAILLARDON[5], (Senior Member, IEEE), AND GIOVANNI DE MICHELI[1]

[1]LSI, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland
[2]Design Group, Synopsys Inc., Sunnyvale, CA 94085, USA
[3]Department of Electrical Engineering and Computer Science (EECS), UC Berkeley, Berkeley, CA 94720, USA
[4]Synopsys, 38330 Grenoble, France
[5]LNIS, The University of Utah, Salt Lake City, UT 84112, USA

Corresponding author: Eleonora Testa (eleanora.testa@synopsys.com)

This work was supported in part by the Swiss National Science Foundation under Grant 200021-169084 MAJesty, in part by the ERC project under Grant H2020-ERC-2014-ADG 669354 CyberCare, in part by the Defense Advanced Research Projects Agency (DARPA) under Grant FA8650-18-2-7849, and in part by Semiconductor Research Corporation (SRC) under Contract 2710.001 and Contract 2867.001.

ABSTRACT In recent years, Boolean methods in logic synthesis have been drawing the attention of EDA researchers due to the continuous push to advance quality of results. Boolean methods require high computational cost, as they rely on complete functional properties of a logic circuit (e.g., don't cares), but usually result in better optimization. In particular, Boolean resubstitution is considered one of the most powerful Boolean methods in logic synthesis. In this paper, we present three novel Boolean resubstitution algorithms designed to be scalable and runtime-effective in a modern synthesis flow. They make use of circuit partitioning techniques and Boolean filtering to be fast and computationally tractable. We also discuss different data structures and reasoning engines, namely truth tables, binary decision diagrams, and satisfiability, that are required to gather don't cares and functional information. As the choice of the engine determines the scalability of Boolean resubstitution we present different scenarios in which the Boolean methods are best driven by one or the other of these. We have implemented the presented resubstitution techniques together with state-of-the-art methods in an industrial logic optimization engine to create a novel resynthesis flow. Our global resynthesis flow achieves significant synthesis results: Within the EPFL synthesis competition, we improve the best-known area results when mapped into LUT-6; when embedded in a commercial EDA flow, the new Boolean resynthesis flow results in 3.12% combinational area savings and 1.34% WNS reduction after physical implementation, at contained (w.r.t. the time of the entire flow) runtime cost.

INDEX TERMS Logic synthesis, Boolean optimization, Boolean resubstitution, area optimization.

I. INTRODUCTION

Electronic Design Automation (EDA) is facing a continuous push to improve *Quality of Results* (QoR). In the logic synthesis field, this has resulted in the development of new synthesis methods, and in the renewed interest for already existing ones. In particular, in light of modern computing capabilities e.g., new SAT-solvers or novel storage means, Boolean resynthesis methods have experienced revived attention in the last few years within industrial and academic flows as demonstrated in the works [1]–[3]. Boolean methods use *don't cares* as degree of freedom for optimization and rely on complete functional properties of Boolean functions. Compared to algebraic methods, which treat a Boolean function as a polynomial, Boolean methods achieve better QoR but imply higher computational cost and consequently have been used cautiously in EDA flows [4]–[6]. Among all Boolean methods, a central role is played by Boolean resubstitution (also called substitution), which, in practice, is considered the most powerful method in terms of QoR. It takes advantage of don't cares [7] and *permissible functions* [8] to express the function of a node using other nodes already present in the

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

logic network to achieve a more compact (with fewer nodes) implementation of the logic network.

In this paper, we propose three novel Boolean resubstitution methods, designed to be scalable in current logic synthesis flows. By scalability, we refer to the runtime complexity of our algorithms. Resubstitution is considered an expensive method w.r.t. runtime and thus filtering techniques (e.g., setting a maximum number of nodes) need to be employed. While this achieves a better scalability, it may overlook many optimization opportunities and result in worse *performance power* and *area* (PPA). The main goal of our work is instead to unleash the full potential of resubstitution by using novel filtering techniques, and by investigating a larger solution space and obtaining improved QoR, at limited (w.r.t. time spent by the synthesis flow) runtime cost.

The first method uses a novel theory of Boolean filtering to reduce the number of gates processed by resubstitution, while still retaining all possible optimization opportunities. The second method revisits the concept of *Maximum Set of Permissible Functions* (MSPF, [8], [9]) to develop so called *Forward Functional Flexibility* (FFF, [1]). FFF is a weaker notion of MSPF, which in practice is able to grasp a good amount of MSPF opportunities at limited runtime cost [10]. Both these methods make use of *windows* to partition the network into smaller subnetworks, to apply our methods efficiently also on large networks. The third method is a novel Boolean resubstitution method based on Boolean difference. Also in this last method, we focus on structural filtering, candidates selection, and partitioning techniques to achieve a scalable resubstitution flow. As compared to our previous works [1], [2],we aim here at giving a generalize version of our techniques and algorithms such that they are independent on the engine chosen to represent Boolean functions and don't cares. Choosing an appropriate data structure and reasoning engine for the implementation determines then not only the applicability but also the scalability of the Boolean resubstitution method. Thus, we also discuss here different reasoning engines and data structures for performing the essential task of detecting permissible functions and don't cares which are the core of all our Boolean resubstitution methods. In particular, we focus on truth tables, *Binary Decision Diagrams* (BDDs, [11]) and *SATisfiability* (SAT, [12]), and we identify practical scenarios, based on circuit characteristics and optimization scope, where Boolean methods are best driven by either one or a blend of them.

The presented resubstitution techniques have been implemented and integrated in an industrial logic optimization engine to create a novel resynthesis flow aiming primarily at area optimization. Our novel global resynthesis flow consists of (i) the proposed Boolean resubstitution methods - implemented and applied using various engines and partition size, (ii) revisited state-of-the-art algorithms, usually involved in modern logic synthesis flows. As far as Boolean resubstitution is concerned, we focus on both improvements to their scalability and the impact of the right choice of the reasoning engine. As a novel result on top of [1], we present here a new

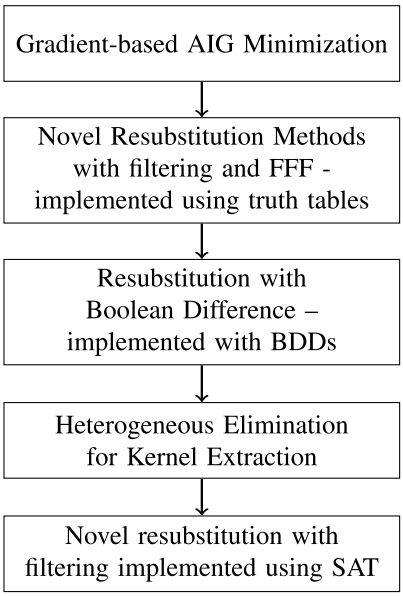


FIGURE 1. Novel complete synthesis flow. It interleaves our resubstitution techniques with state-of-the-art flows as AIG optimization and kerneling. The resubstitution techniques are implemented using different reasoning engines.

implementation of resubstitution that uses SAT as reasoning engine. The improved state-of-the-art methods include (i) a gradient-based *And-Inverter Graphs* (AIGs, [13]) optimization engine, and (ii) heterogeneous elimination and kerneling. The first method improves classical AIG optimization by using a technique that adapts online to apply the most effective AIG transformations, while the second presents an improvement to elimination and kerneling to enhance division and logic sharing to work with heterogeneous thresholds in the same network. An overview of the novel flow – and reasoning engines used for resubstitution – is represented in Fig. 1. Each step will be detailed in the coming sections. As a whole, our global resynthesis flow produces significant synthesis results, both over academic and industrial benchmarks. Concerning the EPFL benchmarks [14], we show significant improvements over the smallest-known AIGs. For instance, we demonstrate 1.3× (26%) size reduction in the smallest-known AIG for the EPFL *voter* benchmark. Further, we also improve the best-known area results in the EPFL synthesis competition, when mapping the AIGs into LUT-6. Regarding industrial benchmarks, our resynthesis flow embedded in a commercial EDA flow for ASICs results in an averaged 3.12% combinational area savings and 2.49% dynamic power reduction, after physical implementation, at limited runtime cost. Even though we target as main goal the area reduction of logic networks, we also enforce a tight control on the number of levels during the industrial synthesis flow, which is known to be correlated with the delay, consequently obtaining an improvement of 1.34% on the *Worst Negative Slack* (WNS) and of 0.82% on the *Total Negative Slack* (TNS) as well. Finally, to evaluate the scalability of our resubstitution methods and the impact of the proposed filtering techniques, we also perform a detailed comparison between

our resubstitution (with and without filtering) and ABC [6] for few test-cases.

The remainder of this paper is organized as follows. Section II provides some background on Boolean methods and don't cares, and Section III details Boolean resubstitution. Section IV proposes three novel Boolean resubstitution methods, while Section V discusses the different reasoning engines required to gather complete functional properties of Boolean functions, which are usually involved in Boolean resubstitution. Section VI shows the complete resynthesis flow and details each step of the flow, while Section VII provides details on all the experimental results. Finally, Section VIII concludes the paper.

II. BACKGROUND ON BOOLEAN LOGIC OPTIMIZATION

Logic optimization methods are divided into algebraic methods and Boolean methods [4], [5], [15]–[17]. Algebraic methods are usually faster and treat a Boolean function as a polynomial. On the other hand, Boolean methods consider the true nature of logic functions by considering Boolean identities and *don't cares* [18]. Don't care conditions relate to the embedding of a Boolean function into the environment, and are usually called *external don't cares*. They consist of both *controllability* and *observability* don't cares. The first is defined as those input patterns that are never produced by the environment, while the latter considers situations when a given output is not observed by the environment. The idea behind Boolean methods is to use the power of Boolean algebra together with the degree of freedom provided by the don't cares to construct local transformations to improve logic networks [4], [8], [19]. For example, due to observability don't cares, the function at a node n may be changed to another function without changing the behavior at the primary outputs. In the transduction method proposed by Muroga [8], [9], this new function is called a *permissible function* for node n, and the set of *all* permissible functions for a node n is its *Maximum Set of Permissible Functions* (MSPF). Consequently to the use of don't cares and Boolean identities, Boolean methods usually achieve better results, but come at higher computational cost and less scalability [4].

Boolean methods have advanced significantly in recent years. Today, different data structures and reasoning engines are available for collecting functional properties and don't cares, and for detecting the existence of permissible functions. As we will discuss next, this choice determines the scalability of the Boolean methods. Here, we give some background on truth tables, BDDs and SAT, together with their use in don't cares computation and logic synthesis. In Section V, we explain in detail the scenarios in which these three produce the best results in driving Boolean resubstitution methods.

A. TRUTH TABLES

A truth table is a canonical representation of a Boolean function where the function values are listed for all possible input combinations. Formally, a truth table is a bitstring of 2^t bits $b_{2^t-1}b_{2^t-2}\dots b_1b_0$, for which $f = b_x$ such that $x = (x_t \dots x_1)_2$ is the integer representation of the input assignment.

Example 1: For instance, the truth table for the 3-input majority function $\langle x_1x_2x_3\rangle$ is given by 11101000. ■

Truth tables have been largely used in logic synthesis to represent and optimize logic functions, as they enable fast computation and equivalence checking of two functions when applied to small windows of logic. As an example, the works in [20] use truth tables as the preferred engine for functional manipulations.

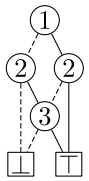


FIGURE 2. BDD for the function $\langle x_1x_2x_3\rangle$. Note that the same function can be represented using the truth table 11101000.

B. BDDs

A BDD [11], [21] is a directed acyclic graph representing a Boolean function. There is one root node, and two leaf nodes (labeled '$\bot$' and '$\top$'). Each internal node (non-leaf node) has a variable associated with it and implements the Shannon expansion of the function w.r.t. a variable x_i: $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$, where f_{x_i} and $f_{\bar{x}_i}$ are the *cofactors* obtained from f when the variable x_i is assigned to `1` or `0`, respectively (i.e., positive and negative cofactors). A *reduced* and *ordered* BDD (called a ROBDD [11]) is a canonical representation for a Boolean function w.r.t. a given variable ordering [11]. If not otherwise specified, in the following we use the term BDD to refer to ROBDD.

Example 2: Fig. 2 shows the BDD for the function $\langle x_1x_2x_3\rangle$. Solid and dashed lines represent positive and negative cofactors respectively, while the '$\bot$' and '$\top$' represent the constant functions `0` and `1` [22]. Each internal node has a variable x_i associated, which is represented by the integer i inside each node. ■

BDDs are largely employed in Boolean optimization methods [2], [4], [23]. BDDs can be used to check quickly if a function is a permissible replacement of another by checking the tautology of their equivalence (note that tautology check is known to be a fast operation when evaluated on BDDs). BDDs are also used for representing and minimizing Boolean relations [23], which are a generalization of don't cares [4] used to capture the flexibility of multi-output circuits. In logic synthesis, BDDs have also been largely exploited for logic function decomposition. The decomposition is obtained by making use of *dominator* nodes, which identify the structures for a particular decomposition type (i.e., AND/OR, XOR, MAJ). As an example, BDS [24] is an optimization system for the synthesis of AND/OR and XOR-based functions, which use a dominator-based decomposition method on BDDs. In [25], an extension to the majority decomposition is illustrated.

C. SAT

SAT solvers have recently been used as Boolean method engine for don't cares computation. This is possible by casting the search for permissible functions into a *SATisfiability problem* (SAT, [12]). A SAT problem takes a formula representing a Boolean function and decides if there is an assignment of the variables for which the function is equal to 1 (satisfiable). Otherwise, the formula is said to be *UNSATisfiable* (UNSAT). In [26] a method to cast don't cares computation into a SAT problem is presented. A miter configuration is proposed, similar to the one used for equivalence checking; a windowing method is also presented to make the SAT solver approach more efficient. We refer the reader to [26] for examples on this method. An updated version of SAT-based don't cares computation has been presented in [27], while, more recently, a SAT-based redundancy removal approach has been described in [28]. More generally, SAT-based logic synthesis techniques are also used in rewriting algorithms. These algorithms optimize a network by replacing small subnetworks with their optimized versions obtained using a SAT-based exact synthesis approach [29], [30].

III. THEORY OF BOOLEAN RESUBSTITUTION

The engines presented in the previous section can be used for gathering functional properties (e.g., don't cares) and verifying permissible functions, and thus for checking the applicability of Boolean transformations. Many different Boolean transformations exist, some examples being *resubstitution*, *rewriting*, *refactoring*, and *redundancy removal*. We refer the interested reader to [4], [5], [31] for a more exhaustive review on Boolean methods, while in this paper we focus on Boolean resubstitution, which, empirically, is well-known to be the most powerful Boolean method in terms of achieved optimization. As an example, consider the optimization script from ABC [6] *resyn2rs*, where major transformations are *rewriting, refactoring, balancing,* and *resubstitution*. When applying the script on the random-control EPFL benchmark *voter* [14], the complete script minimizes the number of nodes by 41%. The same script reduces the same benchmarks of only 25%, when all resubstitution transformations are removed. Many other similar examples can be found, establishing Boolean resubstitution as a powerful technique for logic optimization, employed in many and diverse logic synthesis flows. In the following, we provide some background on Boolean resubstitution, being the core of the proposed synthesis methods.

Boolean resubstitution aims at expressing the function of a node n using other nodes (called *divisors*) already present in the logic network. A transformation is accepted if the new implementation is more compact than the current node implementation, thus leading to size optimization. A k-resubstitution is a generalization of resubstitution, which adds exactly k new nodes and removes l nodes, where l is the number of nodes in the *Maximum Fanout Free Cone* (MFFC, [32], [33]) of n. In this case, size improvement is achieved if $l > k$. In this last scenario, resubstitution adds k new logic operators to the existing logic network. Note that resubstitution techniques are thus usually classified according to the number k of logic operators additionally added, i.e., 0-resubstitution does not add any new operator; 1-resubstitution expresses a logic function by adding one logic operator, and so forth. According to the type of nodes added in the logic network by resubstitution, we also refer to resubstitution as AND-resubstitution, OR-resubstitution, XOR-resubstitution, AND-OR resubstitution, etc.

Due to the use of don't cares, Boolean resubstitution finds more optimization opportunities as compared to algebraic substitution, but it is inherently more expensive [4]. Consider the following example, which shows the use of don't cares for Boolean resubstitution.

Example 3: Consider the logic network [5] with inputs $x_1, \ldots, x_4$ given by

$$\begin{aligned} f &= x_1 \vee (x_2 \wedge x_3 \wedge x_4) \\ g &= x_1 \vee (x_3 \wedge x_4) \end{aligned} \tag{1}$$

where $\wedge$ and $\vee$ represent the AND and OR operators, respectively. We can minimize function f using other nodes in the network, i.e., g by doing:

$$\begin{aligned} f &= x_1 \vee (x_2 \wedge g) \\ g &= x_1 \vee (x_3 \wedge x_4) \end{aligned} \tag{2}$$

where $x_2 \wedge x_3 \wedge x_4$ can be changed into $x_2 \wedge g$ because the minterms where $x_2 \wedge x_3 \wedge x_4$ and $x_2 \wedge g$ differ are in the don't care set. Indeed, $x_2 \wedge x_3 \wedge x_4$ has truth table equal to 1000000010000000 and $x_2 \wedge g$ truth table is 1111000010000000. By considering x_1 truth table 1111111100000000, the two functions have different minterms only when $x_1 = 1$, which is not observable from the output of f because of the OR $\vee$ operation. ■

Different representation of don't cares and varying data structures and reasoning engines have been used in the past years to develop novel and powerful resubstitution methods. For example, in the transduction method proposed by Muroga [8], [9], resubstitution methods were applied by computing permissible functions using truth tables. Unfortunately, at the time, that description was not efficient enough to be applicable to logic networks of reasonable size. On the contrary, modern resubstitution flows as the ones in [1] and in [3] effectively use truth tables for Boolean resubstitution. Permissible functions can also be quickly evaluated using BDDs. The method in [34] uses BDDs and permissible functions to build fast resubstitution techniques. More recently, Miyasaka *et al.* [35] have presented a method that uses a BDD-package without variable re-ordering to accelerate the computation of permissible functions. Concerning SAT-based resubstitution methods, the works in [26], [27] consider SAT-based don't cares computation aiming at resubstitution frameworks.

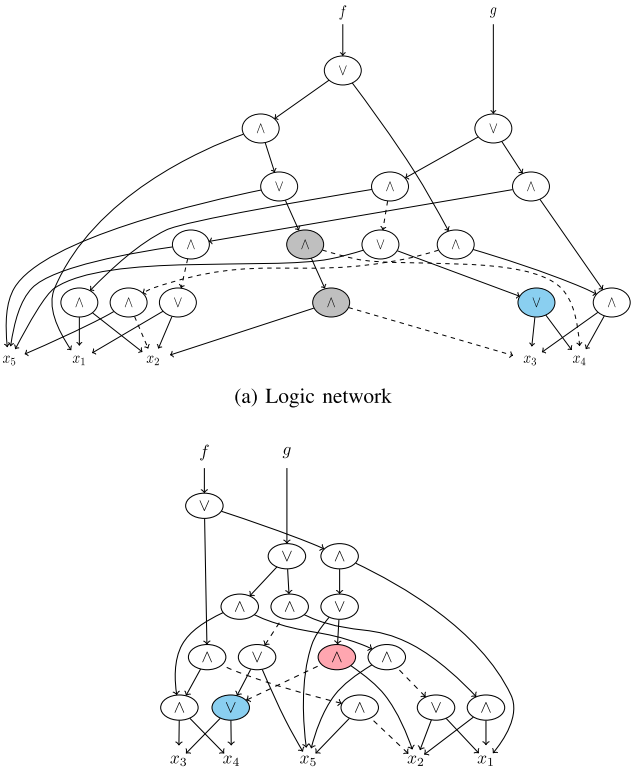
Although Boolean resubstitution achieves better results and is more precise than the algebraic one, it is more expensive at runtime. The right engine selection may improve its

scalability (i.e., runtime complexity), but its application is still limited to small functions. Partitioning and breaking logic networks into smaller subnetworks is thus needed to be able to efficiently compute each node's functionality and to apply Boolean resubstitution in industrial frameworks. An example of this approach is given by ABC [6] resubstitution, that is considered the academic state-of-the-art tool for logic synthesis. Resubstitution within ABC is a technology-independent version of the works in [19], [36] and it is applied to small windows of logic (up to 16 inputs) to contain the runtime complexity. In this scenario, AIGs are used as underlying data structure for k-resubstitution (with k up to 3) and truth tables are used to compute don't cares [33]. Even when resubstitution is applied to small- ($\sim$15 inputs) and medium- ($\sim$20 inputs) size windows of logic, k-resubstitution remains intrinsically expensive due to the high amount of required equivalence checking, which are the primitive operations in all resubstitution algorithms. Consider as an example 1-resubstitution using the 2-input AND gates, applied on a small window of logic with N internal nodes. The goal is to try to express each internal node in the window as the AND of two other nodes in the window. In the worst case $O(N^2)$ equivalence checks are needed for each node, resulting in $O(N^3)$ checks for the whole window. Since linear and sub-linear runtime complexity are desired, Boolean resubstitution is regarded as an expensive method in terms of runtime. For this reason, Boolean filtering techniques are usually employed to strongly reduce the number of candidates for resubstitution, and, at the same time, without losing significant optimization opportunities. Common filtering techniques [1] include, but are not limited to, (i) structural filtering, or (ii) setting a maximum number m of candidates to be tried. Structural filtering comprises, for instance, skipping candidates in the *Transitive FanOut* (TFO) cone of the current node, or skipping nodes whose level is too far away, etc. The best improvement in runtime is although given by setting a maximum number of candidates m, as the total number of equivalence checks decreases to $O(mN^2)$. While this achieves a scalable algorithm, it does not assure a good QoR.

In the next section, we focus on techniques to make our Boolean resubstitution methods scalable, efficient, and applicable to large functions. We discuss novel Boolean filtering techniques, together with a new efficient way for computing permissible functions and don't cares. We also concentrate on partitioning techniques to break the logic into smaller subnetworks. The idea is to increase the solution space spanned by the resubstitution algorithms, but, at the same time, to keep the runtime contained w.r.t. to the time of the synthesis flow.

IV. SCALABLE BOOLEAN RESUBSTITUTION

We discuss here three different scalable Boolean resubstitution methods. First, we present a resubstitution method that uses novel Boolean filtering and windowing techniques to speed up the computation. Then, we describe an improved MSPF calculation, resulting in more efficient don't cares evaluation. Finally, we conclude with a resubstitution method



(a) Logic network

(b) Logic network rewritten using AND-resubstitution

FIGURE 3. Boolean resubstitution example.

based on the Boolean difference. As a remark, the proposed algorithms work on general Boolean networks, in which each internal node represents a Boolean function.

A. RESUBSTITUTION FOR COMPLEX GATES

In this section, we propose a resubstitution algorithm which has the capability of deriving complex gates with efficient runtime. The algorithm makes use of novel Boolean filtering and windowing techniques to develop a novel resubstitution framework. Different types of resubstitution are considered, e.g., AND-operator, OR-operator, 0-resubstitution, etc.

Example 4: Consider as an example the logic network in Fig. 3(a). Each node in Fig. 3 is a 2-input gate, and dashed edges represent inverters. The variables $x_1, x_2, \ldots, x_t$ are the inputs of the network, also called the *support* of the two functions. The subnetwork highlighted in gray in Fig. 3(a) can be rewritten as the AND gate (AND-resubstitution) between the blue node and the input x_2. This results in the new network in Fig. 3(b), which has one fewer node compared to the original network. The pink node is the AND operator added by the resubstitution (1-resubstitution). ■

In the following, first, we discuss the use of Boolean filtering and windowing techniques to speed up the computation, then we present the resubstitution flow.

1) BOOLEAN FILTERING AND WINDOWING

As discussed in Section III, Boolean filtering and windowing play a key role in making Boolean resubstitution techniques scalable and less expensive at runtime. Concerning Boolean

filtering, the runtime complexity remains very high when only structural filtering is applied. Thus, experimentally, setting a maximum number of candidates to be tried is preferred, even if it may overlook advantageous optimization opportunities. In this paper, instead we make use of *canalizing functions* [22], [37] to efficiently implement Boolean filtering rules for resubstitution. Note that the concept of canalizing function is known as controlling values in the EDA community.

A function g is said to be p/q-*canalizing* in x if the cofactor $g_{x=p} = q$ for two constant Boolean values p and q is uniquely determined by assigning x to either 0 or 1. The advantage in using canalizing functions to drive the Boolean filtering for resubstitution is that a whole set of candidate pairs can be excluded by just checking at one single operand. This is summarized in the following theorem:

Theorem 1: Let f be the function of a node to resubstitute using a k-input gate with function $g(x_1, x_2, \ldots, x_k)$ and k candidate nodes that have functions $f_1, f_2, \ldots, f_k$, respectively. If g is p/q-canalizing in x_i, then f_i can only be a valid candidate, if $(f_i(x) = p) \rightarrow (f(x) = q)$, or logically equivalent, if $(f(x) \neq q) \rightarrow (f_i(x) \neq p)$, for all function inputs x. ■

Proof 1: We prove the theorem by contradiction. Assume that $f_i(x) = p$ for some $p \in \{0, 1\}$, but $f(x) \neq q$ for some $q \in \{0, 1\}$. Since g is p/q-canalizing in x_i, we have by definition that $g_{x_i=p} = g_{f_i(x)=p} = q$, which contradicts our assumption. ■

Example 5: The 2-input AND function is 0/0-canalizing in both its inputs. If we want to express some node with function f using the AND of two other nodes with functions a and b, then a can be valid only if $f \rightarrow a$, or equivalently if the check $f \wedge a = f$ is true. ■

Similar checks/rules can be found for many other functions. If a check finds that some node a is not a valid candidate, all combinations in which a is present can be discarded. In our algorithm, AND-resubstitution ($f \wedge a = f$) and OR-resubstitution ($f \vee a = f$) have been easily implemented according to Theorem 1. Note that XOR-resubstitution has also been considered. In this last case, since the XOR-operator is not canalizing, nothing can be concluded by checking at a single operand only, and both operands need to be investigated.

Further resubstitution techniques based on using three operands can also be examined:

Example 6: Consider expressing a function f in terms of three nodes with functions a, b, and c using $f = a \vee (b \wedge c)$. The first step is checking whether a is a valid candidate. If a is invalid, one can continue considering a different node. Otherwise, the validity of b needs to be investigated to filter out candidates for c. When checking b, it is assumed that $a = 0$, since otherwise the equation is satisfied independently from b. Therefore, Theorem 1 can be applied similarly by considering $f_{a=0} = b \wedge c$, which is 0/0-canalizing in b. ■

The following three-operands resubstitutions have been taken into account: AND-OR-resubstitution ($f = a \vee (b \wedge c)$), OR-AND-resubstitution ($f = a \wedge (b \vee c)$), XOR-AND-resubstitution ($f = a \wedge (b \oplus c)$) and XOR-OR-resubstitution ($f = a \vee (b \oplus c)$). We also consider a MUX-resubstitution ($f = (a \wedge b) \vee (\bar{a} \wedge c)$).

Apart from using the mentioned Boolean filtering, the resubstitution flow is made scalable due to its application to small/medium-size windows of logic created around a node. This is a common practice in both academic and industrial tools to make resubstitution applicable to larger benchmarks. The window is built using a procedure inspired from [38]: For each node n, first, a *convergent cut* is found, then this is expanded by adding external nodes that are not contained by the cut, but have fanins inside the cut. The procedure is iterated until a volume limit is hit or no more nodes can be added. The final result is a window, having the same number of inputs as the original cut, but with more outputs.

2) RESUBSTITUTION FLOW FOR COMPLEX GATES

Alg. 1 depicts the pseudocode for the resubstitution algorithm for complex gates based on Boolean filtering and windowing. Boolean filtering allows us to only pay attention to profitable transformations, and thus to save runtime. Consider for instance the *voter* benchmark of the EPFL suite [14]. The ABC command *resub -K 10 -N 1* [6] reduces the size by about 14% and takes around 0.09 seconds. The same command run with $N = 2, 3$ (i.e., 2 and 3 resubstitution), improves the size of 16% and 20% with a runtime of 0.13 and 0.21 seconds, respectively. Our resubstitution technique with Boolean filtering reduces instead the original size by 26% in 0.1 seconds.[1] A maximum number of candidates is still used, even though rarely hit, to keep runtime under control for corner cases. Alg. 1 considers each node in topological order, but the resubstitution is applied only to a small window of logic. The window is obtained by expanding a convergent cut into a window as previously discussed. Additional filters are used (e.g., see line 8-9) to ignore windows that are too thin, because they are unlikely to lead to any advantageous resubstitution. Different types of resubstitution are tried for each node (see lines 13–36 in Alg. 1), and the first successful one is kept (i.e., waterfall model). The variable *nresub* controls the computational complexity of the algorithm, as it sets how many different types of resubstitution are tried; practically, we can set it to the maximum value for a variable of type int (*max-int*) so that all types of resubstitution are attempted. For the *zero-resub* case only equivalent gates in the window are combined, up to complementation. Moreover, only candidates with the same support as n are tried. If *zero-resub* is successful, no other resubstitutions are tried for the same node and the loop moves to the next node in the window. In the negative case, other types of resubstitutions are tried. For the resubstitution moves, the savings of each

[1] Our size improvement is measured after AIG structural hashing (strashing) for the sake of fair comparison with ABC. Note also that truth tables have been used in this example to retrieve functional properties of nodes. This result is the one reported in Table 2 for the *voter* benchmark.

Algorithm 1 Resubstitution for Complex Gates

Input: Logic network, cut-size, filter-volume, nresub, zero-gain (zg), max-number-of-candidates (mc)
Output: Resynthesized logic network

```
list ← topological-sort-network(network)
for each node m in list do
    if node is not a MFFC root then
        continue
    end if
    cut ← find-reconvergent-cut(m, cut-size)
    expand-cut-into-window(cut)
    if volume cut / size cut < filter-volume then
        continue
    end if
    wdw ← topological-sort-network(window)
    for node n in wdw do
        if (nresub > −1) && zero-resub(n, window) then
            continue
        end if
        if (nresub > 0) && and-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 1) && xor-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 2) && ao-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 3) && xa-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 4) && ax-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 5) && mux-resub(n, wdw, zg, mc) then
            continue
        end if
        if (nresub > 6) && mx-resub(n, wdw, zg, mc) then
            continue
        end if
    end for
end for
network-cleanup-and-sweeping(network)
```

move are always checked: If the number of extra nodes added by resubstitution is greater than or equal to the sum of the MFFC size and the zero-gain variable *zg*, the loop moves directly to the next node because no size saving is possible. The supported resubstitutions – in increasing computational complexity order – are: *and* (2-input and node), *xor* (2-input xor node), *ao* (and-or nodes), *xa* (xor-and nodes), *ax* (and-xor nodes), *mux* (mux node) and *mx* (mux-xor nodes). Inside each of these resubstitution rules, discussed Boolean filtering methods as the ones based on canalizing functions are used to accelerate the computation (and save runtime). Also, a maximum number of nodes – here called *mc* – to be tried as candidates for the resubstitution procedure can be set. More details about the algorithm can be found in [1]: For instance, the saving can be made more accurate if the logic network is mapped, so that real area savings can be measured instead of number of nodes.

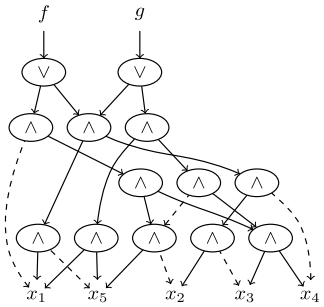


FIGURE 4. Boolean resubstitution with don't cares applied on the network from Fig. 3(a). More optimization opportunities are found as compared to Example 4.

B. RESUBSTITUTION WITH FORWARD FUNCTIONAL FLEXIBILITY

In this section, we present another resubstitution flow which makes use of don't cares expressed as permissible functions. Note that the MSPF is one of the most powerful interpretations of don't cares in logic synthesis. However, its computation can be very time-consuming, especially when a logic network has many convergent paths [8]. Here, we describe a second resubstitution method which makes use of a weaker notion of MSPF, called the *Forward Functional Flexibility* (FFF) information [1], which helps in accelerating the computation of permissible functions. It is worth noting that optimization opportunities and QoR are improved when Boolean resubstitution is enriched with don't cares information.

Example 7: Consider the logic network from Fig. 3(a). Resubstitution with external don't cares results in the logic network in Fig. 4, which has 3 fewer nodes as compared to the original network. The number of levels is also optimized, reducing them from 5 to 4. The improvement is larger compared to Example 4, as more resubstitution opportunities are found. ■

In the following, first, we give the background on FFF, then we illustrate the resubstitution flow.

1) FORWARD FUNCTIONAL FLEXIBILITY

The FFF is a weaker notion of the MSPF that helps improving the runtime, and overcomes the difficulties in computing the MSPF of gates and connections each time that the network is optimized/updated [8], [9]. FFF considers only the permissible functions generated by the *forward propagation* of a node's functionality and don't cares to its TFO. In contrast, the full MSPF considers *all* possible permissible functions by definition, thus also those originated by the interaction of a node with its *Transitive FanIn* (TFI) and the rest of the network. Practically, FFF grasps a good amount of MSPF opportunities but still fits in a tight runtime budget [10].

We refer to FFF(C, n) as the FFF computed for a specific node n in a small or medium network C. The FFF(C, n) information can be interpreted as follows: For a given input

Algorithm 2 Forward Functional Flexibility Computation

Input: Logic network C, Current node n
Output: Forward Functional Flexibility FFF(C, n)

```
1: DC(n) ← logic constant 1
2: FFF(C, n) ← logic constant 1
3: for each node m ∈ TFO(n) in topological order do
4:     for each fanin k of m do
5:         i = 0
6:         if k has DC info then
7:             DC-in(i) ← DC(n, k)
8:         else
9:             DC-in(i) ← logic constant 0
10:        end if
11:        i++
12:    end for
13:    SOP ← get-sop-representation(m)
14:    DC(n, m) ← logic constant 0
15:    acc-sop ← logic constant 0
16:    for each term of the SOP do
17:        term ← logic constant 1
18:        flex-term ← logic constant 0
19:        for each literal of the term do
20:            index ← literal_index
21:            flex-term ← dc-and(literal, DC-in(index), term, flex-term)
22:            term ← literal ∧ term
23:        end for
24:        DC(n, m) ← dc-or(term, flex-term, acc-sop, DC(n, m))
25:        acc-sop ← acc-sop ∨ term;
26:    end for
27:    if m ∈ PrimaryOutputs(C) then
28:        FFF(C, n) ← FFF(C, n) ∧ DC̄(n, m)
29:    end if
30: end for
31: return FFF(C, n)
```

combination, if the value of the FFF of n is 1, then the value of n can be flipped without effecting any of the primary outputs in C. Consequently, when checking candidates for resubstitution, it is thus sufficient to compare the node functions only at their non-flexible input combinations. Let n_1 and n_2 be two nodes in a network C, representing the functions f_1 and f_2 respectively, then, we can consider n_1 and n_2 equal in C, if

$$\left(f_1 \wedge \overline{\mathrm{FFF}}(C, n_1)\right) = \left(f_2 \wedge \overline{\mathrm{FFF}}(C, n_2)\right).$$

where the $\bar{}$ is the complementation of the function.

The pseudocode to compute FFF(C, n) for all nodes is depicted in Alg. 2. It initially assigns maximum flexibility to the global FFF(C, n) (logic constant 1), and it also set the local don't care (*DC* in Alg. 2) for n to logic constant 1. Note that in Alg. 2, the DC(n, m) for some node m w.r.t. n has an opposite meaning than the FFF. That is, if the don't care of node m w.r.t. n for a given inputs combination i is 0, it means that node m at index i is not sensible at flipping the value of node n at index i.

In Alg. 2, the DC is propagated from n to the primary outputs in topological order. For each node in the TFO of n, the DC is computed by considering the *Sum Of Products* (SOP) representation; in particular, while parsing the SOP, AND/OR operators are replaced by special operators,

Algorithm 3 2-Input AND Function to Include Don't Cares Information

Input: a, $DC(a)$, b, $DC(b)$
Output: $dc\_and(a, DC(a), b, DC(b))$

```
1: aux1 = (a ∨ DC(a)) ∧ DC(b)
2: aux2 = (b ∨ DC(b)) ∧ DC(a)
3: res = aux1 ∨ aux2
4: return res
```

Algorithm 4 2-Input OR Function to Include Don't Cares Information

Input: a, $DC(a)$, b, $DC(b)$
Output: $dc\_or(a, DC(a), b, DC(b))$

```
1: aux1 = (ā ∧ DC(a)‾) ∧ DC(b)
2: aux2 = (b̄ ∧ DC(b)‾) ∧ DC(a)
3: res = aux1 ∨ aux2
4: return res
```

called *dc_and* and *dc_or*, which include the local don't cares information for each operand. The two algorithms used to include the DC information in the AND and OR operators are depicted in Alg. 3 and 4, respectively. At each primary output m, the FFF(C, n) is updated by and-ing itself with the complement of DC(n, m): This is needed as only the common intersection between the FFF of all outputs can be safely used for optimization purposes. The algorithm stops when all primary outputs have been processed.

Note that the FFF techniques can also be used to further enhance the previously presented Boolean filtering techniques; for instance, the validity check for node a (see Example 5) when incorporating functional flexibility, is given by:

$$\left(f \wedge \overline{\mathrm{FFF}}(C, f)\right) \rightarrow \left(a \wedge \overline{\mathrm{FFF}}(C, a)\right)$$

2) RESUBSTITUTION FLOW WITH FORWARD FUNCTIONAL FLEXIBILITY

The global resubstitution flow which uses FFF is shown in Alg. 5. The idea for resubstitution with FFF is to update the information for the current node n in the circuit, such that resubstitution moves can take advantage of the don't cares flexibility. The procedure remains identical to Alg. 1, but with updates inside the window processing. Note that the same window computation described in IV-A is used together with FFF to optimize the Boolean flow and make the method more scalable. The algorithm stops (i.e., we do not try any other node) in case the resubstitution is successful. This is necessary to preserve the correct functionality as node functions may be modified by the use of don't cares. Alternative approaches to this window-skipping have been considered, where FFFs are incrementally updated after every successful resubstitution. Experimentally, window-skipping is more advantageous when considering a tradeoff between

Algorithm 5 Resubstitution With *Forward Functional Flexibility*

Input: Logic network, cut-size, filter-volume, nresub, zero-gain, max-candidates
Output: Resynthesized logic network using FFF information

```
list ← topological-sort-network(network)
for each node m in list do
    if m is not a MFFC root then
        continue
    end if
    cut ← find-reconvergent-cut(m, cut-size)
    expand-cut-into-window(cut)
    if volume cut / size cut < filter-volume then
        continue
    end if
    wdw ← topological-sort-network(window)
    for each node n in wdw do
        FFF ← compute-FFF(n, wdw)
        update-information-with-flexibility(n, FFF)
        run-resub-flow()
        if resub with FFF is successful then
            break
        end if
    end for
end for
network-cleanup-and-sweeping(network)
```

QoR and runtime, as it allows to explore more don't care combinations.

C. RESUBSTITUTION BASED ON BOOLEAN DIFFERENCE

Here, we present a Boolean resubstitution flow based on Boolean difference, described for the first time in [2]. The *Boolean difference* [4] of a Boolean function $f(x_1, x_2, \ldots, x_t)$ w.r.t. an input variable x_i is defined as:

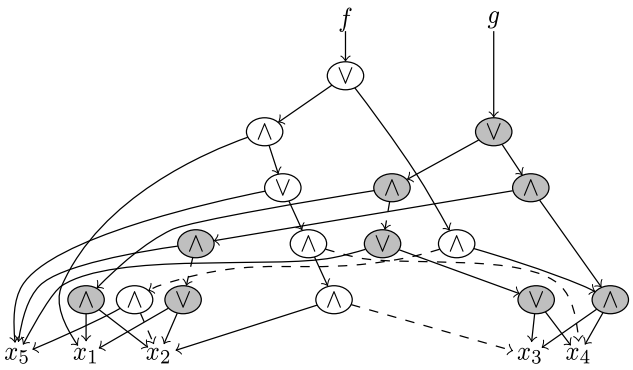
$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i} \tag{3}$$

where f_{x_i} and $f_{\bar{x}_i}$ are the two cofactors and $\oplus$ is the XOR operator. It states whether function f is sensitive to any change in input x_i. In a similar way, the Boolean *difference* of two Boolean functions $f(x_1, x_2, \ldots, x_t)$ and $g(x_1, x_2, \ldots, x_t)$ is defined as:

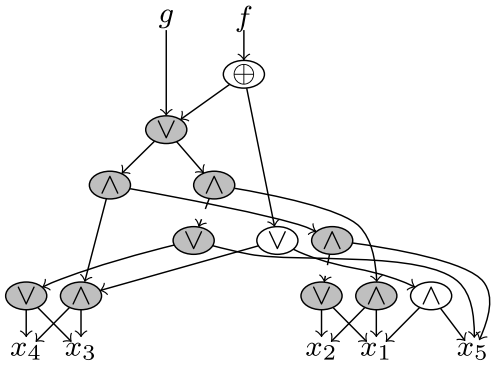
$$\frac{\partial f}{\partial g} = f \oplus g, \tag{4}$$

It indicates whether the two functions are functionally equivalent (i.e., the Boolean difference value w.r.t. inputs assignments is 0) or not (i.e., they have a Boolean difference equal to 1).

According to Boolean difference, each function f can be written as $f = \frac{\partial f}{\partial g} \oplus g$. The main advantage of the resubstitution method comes from the synthesis of the Boolean difference $\frac{\partial f}{\partial g}$, as g is a node already in the logic network. A small Boolean difference implementation could result in size optimization for the logic network. Note that this method



(a) Logic network for functions f and g (in gray)



(b) Function f rewritten as $f = \frac{\partial f}{\partial g} \oplus g$

FIGURE 5. Boolean difference example.

mainly addresses XOR-rich circuits and is thus responsible for revealing further XOR optimizations that are instead not found by generalized resubstitution methods.

Example 8: Consider as an example the logic network for function f and g in Fig. 5(a). Function f is rewritten as $\frac{\partial f}{\partial g} \oplus g$ in Fig. 5(b). The function g is the one highlighted in gray in both Fig. 5(a) and (b). The small size of the Boolean difference network results in a decreased total number of nodes (from 16 to 12). ■

We refer to function f and g as *candidates* for Boolean difference, and to the inputs variables $x_1, x_2, \ldots, x_t$ as their support. We use f and g both for the corresponding nodes in the logic network and for the function they represent.

In the following, we first present the selection of the candidates and their support, and we discuss algorithms to compute the Boolean difference between two nodes. Then, the Boolean difference resubstitution flow is presented.

1) CANDIDATES & BOOLEAN DIFFERENCE COMPUTATION

To ensure the scalability of this Boolean method, we evaluate and apply the Boolean difference locally on limited size circuit partitions, similarly to the two previously described methods. The partitions are created by first collecting all the nodes in topological order, and then by sorting them according to the similarity in their structural support. Each partition respects some decided characteristics, e.g., maxi-

Algorithm 6 Boolean Difference Implementation

Input: Two nodes f and g, xor_cost, all_functions
Output: A new node boolean_diff equal to $\frac{\partial f}{\partial g} \oplus g$

```
1:  boolean_diff ← 0
2:  b_diff ← f ⊕ g
3:  if b_diff already exists in all_functions() then
4:      return corresponding node
5:  end if
6:  if (cost(b_diff) > threshold) then
7:      return null
8:  end if
9:  saving← MFFC(f)-size + nodes_sharing
10: if (size(b_diff) + xor_cost > saving) then
11:     return null
12: end if
13: boolean_diff ←b_diff ⊕g
14: return boolean_diff
```

Algorithm 7 Resubstitution Flow Based on Boolean Difference

Input: Logic network, xor_cost
Output: Resynthesized logic network using the Boolean difference

```
1:  lists ←topological-sort-partitions(network)
2:  for each list in lists do
3:      all_functions ←functions for all nodes in list
4:      for nodes f in list do
5:          for nodes g in list do
6:              if f = g then
7:                  continue
8:              end if
9:              if f and g are not good candidates then
10:                 continue
11:             end if
12:             diff← Boolean_difference(f, g, xor_cost, all_functions)
13:             if size(diff) ≤ size(f) then
14:                 Change f with diff in network
15:             end if
16:         end for
17:     end for
18: end for
19: network-cleanup-and-sweeping(network)
```

mum number of primary inputs, maximum number of internal nodes N, maximum number of levels, etc. In practice, we give priority to the limit on the maximum number of levels, as they correlate with the complexity of the reasoning engine (i.e., BDDs) selected for the Boolean difference computation. Nevertheless, we also ensure partitions to have a limited number of primary inputs and a limited size. Experimentally, promising bounds on the number of levels range from 5 to 30, with maximum size of 1000 nodes.

To find suitable candidates f and g, all pairs of nodes inside each partition are considered. The primary inputs of the partition are the supports for the computation. In this case, the time complexity of the resubstitution is quadratic w.r.t. the partition size N in the worst case. We thus (i) fix the maximum number m of pairs to be tried, (ii) apply all structural and functional filtering described in [2].

The pseudocode to compute the Boolean difference between two nodes in the same partition is presented in Alg. 6. The hashtable *all_functions* stores the pre-computed functions for all nodes in the partition, and the Boolean difference is computed as XOR between the two functions f and g. If the function of the Boolean difference already exists in the hashtable, the corresponding node is returned.

To make the computation faster, we apply structural filtering on the reasoning engine used to store the Boolean functions. For instance, when BDDs are used, the structural filter skips pairs of nodes for which the BDD does not meet the given size requirements. For this case, the algorithm returns *null*. We skip nodes according to two main criteria, being (i) the size or cost of the reasoning engine (here called *cost(b_diff)*), (ii) a large network implementation. The first constraint consequently ensures a limited size implementation for the Boolean difference, but it may overlook some optimization opportunities. The savings are evaluated as the sum of sharing of nodes between the Boolean difference implementation and the existing network, and the size of the MFFC of f. The algorithm concludes with the implementation of the Boolean difference node (line 13 in Alg. 6) as an AIG, obtained by strashing the corresponding network, followed by optimization algorithms from the state-of-the-art to guarantee an optimized implementation. The *xor_cost* is the number of AIG nodes needed to implement the functionality of a two-input XOR. In our implementation, the *xor_cost* is a parameter that takes into account the specific technology involved, in which the XOR node may have a different area ratio as compared to the AND/OR nodes.

2) RESUBSTITUTION FLOW BASED ON THE BOOLEAN DIFFERENCE

Alg. 7 depicts the pseudocode of the Boolean difference-based resubstitution flow, applying the resubstitution to each partition of the entire network. The partitions can be chosen to be separate or overlapping to cover more optimization opportunities. The algorithm precomputes all functions in the hashtable, and considers all nodes in topological order. Trivial pairs of nodes are skipped according to criteria discussed in Section IV-C1. Alg. 6 is used to obtain the new implementation of f using the Boolean difference. A new implementation of f is accepted only if (i) it leads to size reduction, and (ii) it does not increase the number of nodes. The latter could reshape the network, opening new optimization opportunities and helping to escape local minimum.

As a final remark, it is worth mentioning that all the Boolean resubstitution methods presented are independent on the data structure and reasoning engine (i.e., truth tables, BDD or SAT) used to compute the node functionalities and the don't cares. The right choice of the engine can determine both the scalability and the QoR of a given method. In the next section, we discuss in detail the selection of the right

data structure or reasoning engine depending on the circuit's characteristics and the optimization opportunities.

V. TRUTH TABLES, BDDs, OR SAT?

Boolean methods are based on functional properties of a logic circuit, as don't care information. To gather such functional properties, expensive logic data structures and reasoning engines are required, such as truth tables, BDDs, and SAT. In the last two decades, improvements in SAT solving have made SAT-based methods more scalable than those based on BDDs and truth table, which consequently grew outdated. While we acknowledge the advantages of SAT-based methods, it appears in practice that there are still several synthesis scenarios in which BDDs or truth tables are preferable to SAT, in terms of QoR and/or runtime.

This section discusses how to identify scenarios, based on circuit characteristics and optimization scope, where Boolean methods (and Boolean resubstitution) are best driven by either truth tables, BDDs, SAT, or a combination of these three.

A. TRUTH TABLES

Truth tables are efficiently stored in computers as a concatenation of words. Boolean functions with t variables require 2^{t-k} words, where $k = \log_2(\text{word-size})$. It follows each 64-bit (32-bit) word can store a 6(5)-input truth table. For circuits or sub-circuits having less than 16 inputs, truth tables are remarkably fast to compute in practice, as they have low memory footprint and no formulation overhead. Furthermore, truth table computation may be parallelized w.r.t. words and distributed over different threads. For example, 64-bit words operating with a 16-input truth table require bit-level operations among 1024 (independent) words. Distributing such computation over 16 threads, which is common in EDA applications, reduces the latency bottleneck to just 64 consecutive bit-level word operations.

As of today, functional properties of circuits up to 16 inputs are most efficiently computed via truth tables. The overhead of formulating and solving a SAT problem, or handling a BDD manager for the same circuit usually takes considerable amount of runtime.

Example 9: Consider an XOR-rich parity circuit over 16 variables, with many functionally identical nodes originating from partial SOP collapsing during synthesis. Depending on the depth of XOR collapsing, the circuit size can grow over several thousands of nodes. In our case, we deal with about $10k$ AIG nodes. Assume the goal is to merge all functionally equivalent nodes, up to complementation, in this circuit. If the task is performed using truth tables, it takes about 1 second of runtime. When using a SAT-based formulation of the problem, instead, it takes more than 2 minutes to obtain the same result. BDD-based methods take tens of seconds, ranging between 15 seconds and 30 seconds, depending on the settings for static and dynamic variable re-ordering. ■

B. BDDs

BDDs are a compact canonical representation form. Compared to truth tables, which are also canonical but always exponential-sized, BDDs allow polynomial-size for many functions and variable orderings of practical interest [39]. However, other functions, such as multiplication and *hidden-weighted bit*, have exponential-size BDDs for any variable order [39].

When dealing with a medium-large function, whose exact properties are unknown, BDDs construction time can differ sensibly. Empirically, BDDs are always built rapidly, compared to truth tables, for the following circuit cases:

1) Circuits with less than 20 primary inputs,
2) Circuits that, if decomposed into an AIG, have depth $d < 20$ levels,
3) Circuits with a small internal nodes over primary inputs ratio, i.e., volume $v < 2$,
4) Circuits with special properties that facilitate BDD construction, e.g., symmetries.

For the cases above, where the primary inputs are less than 16 and truth tables are not desirable, BDDs are the fastest alternative in the majority of cases. It is worth noting that corner cases for points 3) and 4) exist, i.e., ripple carry adders whose BDDs are built with bad variable order, or symmetric functions with many variables, etc. However, these cases represent a small fraction of the ones encountered in practice and can be detected with some extra filtering.

Example 10: Consider Boolean resubstitution over the MCNC circuit $k2$ [40]. After decomposing it into an AIG, with light sharing, this circuit shows 20 levels, 2580 nodes, and 45/45 inputs/outputs. Building a BDD for this circuit takes less than 10 *ms* with a modern BDD package. Such BDD can then be used for implementing the Boolean filtering rules described Section IV-A, in the same way as they would be implemented with truth tables. Using SAT for each resubstitution move, spanning the whole circuit, would result in notable runtime overhead. ■

C. SAT

SAT solving provides answers to Boolean decision problems. Many Boolean tasks in synthesis can be formulated as decision problems, e.g., *"is this portion of logic redundant?"* Advances in SAT solving have made very large problems solvable in a relatively small amount of runtime [12], consequently, SAT is very appealing in logic synthesis and other fields in EDA. It is fair mentioning that there is some overhead in translating circuit-SAT problems in *Conjunctive Normal Form* (CNF), which is the standard form for solving SAT. Also, SAT provides less information content w.r.t. BDDs or truth tables, which instead encapsulate complete functional information. Nevertheless, SAT is the preferred engine for Boolean methods in the following cases:

1) The Boolean method is global and can be formulated as a decision problem,

2) Implicit enumeration and pruning of large search spaces is required, which is efficiently done by modern SAT solvers,
3) The circuit, or sub-circuit, does not fit with the previous truth tables and BDD preferable cases.

Example 11: Consider redundancy removal over the EPFL circuit *mem_ctrl* [14]. After decomposing it into an AIG, with light sharing, this circuit shows 115 levels, 46k nodes and 1205/1231 input/outputs. Given its size, it is apparent that finding global redundancies via truth tables or BDDs is not feasible. On the other hand, redundancy removal based on SAT formulation [28] and solving is capable to process this benchmark in about 30 seconds, removing 11.8k redundant nodes in the AIG. ■

Based on the proposed guidelines, we instrument our Boolean resubstitution methods to choose between truth tables, BDDs or SAT engines to derive, in the most efficient way possible, the (sub)circuit properties necessary for optimization.

VI. GLOBAL RESYNTHESIS FLOW

In this section, we detail our global resynthesis flow. We have integrated the presented resubstitution techniques in an industrial logic optimization engine, together with revisited state-of-the-art methods which are usually involved in modern logic synthesis flows. We created a global Boolean resynthesis flow which runs the following optimizations (see Fig. 1):

1) *Gradient-Based AIG Minimization*, which revisits classical AIG optimization by using a technique that adapts online to apply the most effective AIG transformations [2];

2) *Novel Resubstitution Methods*, which includes our novel resubstitution techniques with Boolean filtering, windowing, and FFF implemented using truth tables (see Alg. 1 and 5);

3) *Resubstitution with Boolean Difference*, which consists of our novel resubstitution flow based on Boolean difference from Alg. 7;

4) *Heterogeneous Elimination for Kernel Extraction* to enhance division and logic sharing to work on heterogeneous thresholds within the same network [2];

5) *SAT Resubstitution*, which includes our novel resubstitution techniques with Boolean filtering and windowing, implemented using SAT.

In the following, we describe each step used in the proposed flow. For the Boolean resubstitution methods, we focus on the data structure or reasoning engines and the size of the partitions. Concerning the revisited state-of-the-art methods, we concentrate on their improvements compared to the state-of-the-art. Note that this Boolean resynthesis flow may produce better results when iterated multiple times, e.g., 2 to 5 times, depending on the specific runtime budget.

A. GRADIENT-BASED AIG MINIMIZATION

This step implements a method [2] which presents improvements to classical AIG minimization by (i) using gradient-based decomposition, and (ii) exploiting circuit partitioning techniques. AIG optimization traditionally consists of a *script* [6], which is a predetermined sequence of optimization techniques, homogeneously applied to the whole network, e.g., *resyn2rs* from ABC. While a recent work [41] have applied machine learning to find better sequences of primitive optimization techniques, we follow a different strategy, aiming at making AIG optimization automatically *adaptive* and *diverse*. Being *adaptive*, we learn on-the-fly what are the most effective AIG transformations, using gradient computation of the gain, and consequently modifying the next attempted transformations online. Being *diverse*, we try different types of AIG transformations on the same region of logic and making results compete *locally* rather than *globally*. We select the best result either in parallel or in a waterfall model.

Different moves are considered, e.g., *rewriting*, *refactoring*, available in low and high effort modes, trading runtime for QoR. All moves have an associated cost, which correlates to their runtime complexity. The gradient-based AIG engine runs together with a partitioning engine of different sizes depending on the intended scope of the optimization. First, unit cost moves are tried for each partition and are iterated while $gain > 0$.[2] When a local minimum is reached, the gain value switches to 0 and consequently higher cost moves are introduced in the AIG engine. Recording the most successful moves and their sequence is done on-the-fly w.r.t. partitions and network structure. Consequently, moves with high success on the current design can be tried with higher priority in the next iterations.

The number of moves to be tried is decided by a *cost budget* that can be automatically increased by the AIG engine, if the gain gradient exceeds a specific threshold over the last k iterations. This means the AIG engine continues optimizing a logic network if the improvement trend is satisfying. On the other hand, if the gain gradient is 0 over the last k iterations, the AIG engine can terminate beforehand. Both the k factor and gain thresholds can be specified when running the AIG engine. In our experiments, a cost budget equal to 100 and $k = 20$ with minimum gain gradient equal to 3%, provides some of the best AIG optimizations seen over academic and industrial benchmarks [2].

B. NOVEL RESUBSTITUTION METHODS

This step implements our novel resubstitution flow, which uses Boolean filtering, windowing, and FFF to speed up the computation. These are the resubstitution methods presented in Alg. 1 and 5. In this scenario, partitions of small-size (max. 15 inputs) have been considered, and truth tables have been selected as data structure. Efficient bitwise manipulation—for instance, the one discussed in [42]—allows fast calculation of truth tables, functional support, etc. for all nodes in the window. Note that, equivalently, BDDs could be used for the evaluation of permissible functions and MSPF, as in the method presented in [2].

[2]All moves are designed to have $gain \geq 0$ at all times, otherwise the corresponding change is reverted.

C. RESUBSTITUTION WITH BOOLEAN DIFFERENCE

This step implements the Boolean difference-based resubstitution from Alg. 7. The Boolean difference method is applied to medium-size partitions obtained as discussed in Section IV-C1. Thanks to the contained size of the partitions, BDDs allow fast Boolean difference computation and are chosen as engine for the computation. We do not perform any variable ordering for the BDDs, to both save runtime and because we are dealing with small BDDs. This makes a higher amount of memory needed by the BDD package and, thus memory plays a crucial role in the Boolean difference computation. For instance, for the EPFL *cavlc* benchmark [14], the algorithm does not converge in a reasonable amount of time unless the memory used for the BDD of the difference is freed at each iteration. In this last case, the algorithm is applied to the whole network, which has 10 inputs and more than 600 nodes. We thus set a memory limit for the BDD package to prevent any memory issue. If this maximum memory limit is hit during the algorithm, the computation is bailed out and it results in a BDD of size 0 for the given node. This node is disregarded in the next steps.

The cost for the structural filtering in Alg. 6 is set by the size of the BDD. To have good QoR and feasible runtime, we found an empirical value of 10 to be a suitable tradeoff for the threshold. The second filter in Alg. 6 skips nodes whose *savings* is smaller than the empirical threshold set by the BDD size and the *xor_cost*. Limiting the size of the BDD of the Boolean difference sets an upper bound on the number of AIG nodes to implement the Boolean difference. Note also that thanks to the use of BDDs, information for functional filtering of pairs are directly available. The final circuit is obtained by strashing the BDD network into an AIG.

D. HETEROGENEOUS ELIMINATION FOR KERNEL EXTRACTION

This method enhances the state-of-the-art *elimination - kernel extraction* [15], which is one of the most effective techniques in logic optimization, as it can share large portions of logic circuits that are hard to find with other methods. For example, kernel extraction is capable of identifying common factors between large (hundreds to thousands of inputs) operators appearing in HDL descriptions of decoders and control logic.

The effectiveness of kernel extraction stands on the properties and characteristics of the node SOPs. Before kernel extraction, node elimination[3] is often used to create larger SOPs, keeping under control the maximum number of terms or literals, to enable more extraction opportunities to be found. However, elimination is usually run homogeneously on the network, i.e., with the same thresholds on the maximum number of terms or literals. This means that the resulting SOPs have similar size—but not similar characteristics—which is where the extraction opportunities arise.

[3]Node elimination, also known as forward node collapsing, aims at collapsing a node into its fanouts' SOPs. As a result, the node is eliminated.

In this paper, we take advantage of partitions, whose computation can be distributed in parallel, to enhance elimination and kernel extraction to work on heterogeneous thresholds within the same network. The idea is that elimination produces different QoR according to the type of circuit. For instance, it produces good results for circuits with large SOPs size. In this case, a large threshold is thus preferable. On the other hand, a small threshold should be used for circuits containing parity and linear functions. First, partitions of the network are created, then elimination and kernel extraction is applied to each partition with different eliminate thresholds. We only keep the best result, which is the one reducing the largest number of literals. The elimination process within each partition works as follows: First, for each node, the variation in the number of literals that would result from the collapsing of the node into its fanouts is evaluated. If this variation is less than the determined threshold, the collapsing is performed. The operation is repeated until no node gets collapsed into its fanouts. In practice, we tried the following eliminate thresholds: (-1, 2, 5, 20, 50, 100, 200, 300).

E. SAT RESUBSTITUTION

The step implements our novel resubstitution flow based on Boolean filtering and windowing (see Alg. 1), when SAT is used as underlying reasoning engine. In particular, the Boolean filtering checks (based on canalizing functions) described in Section IV-A1 are implemented using a SAT encoding and solving. After generating a SAT formulation of the problem, the SAT solver is used to check the resubstitution opportunity. If the SAT problem is UNSAT, it means the Boolean filtering checks are successful and the divisor is a valid candidate for resubstitution. On the other side, if the result is SAT, the Boolean filtering rules are not passed and the resubstitution is not advantageous. In this case, a counterexample is produced by the SAT solver that can be used to refine the simulation for filtering. For this algorithm, the saving in Alg. 1 was made more accurate by using mapped logic networks as proposed in [1].

Altogether, the proposed synthesis flow enables significant synthesis results over both academic and industrial benchmarks, when addressing size optimization. These results are presented in the next section.

VII. EXPERIMENTAL RESULTS

In this section, we evaluate the efficacy of our proposed global resynthesis flow from Section VI. Note that after each transformation, the logic network is transformed into an AIG to have a consistent interface and cost among the various steps of the flow. Within modern EDA flows, Boolean methods are usually called during logic structuring, which mainly aims at reducing area. We thus target as main goal size/area reduction of logic networks; nevertheless, within the industrial flow, we also enforce a tight control on the number of nets and the number of levels during synthesis, as this correlates with delay and congestion in the industrial flow. In the following, we first present results on academic benchmarks, then we

TABLE 1. Best 6-LUT area results for the EPFL benchmarks.

Benchmark	I/O	Previous Best Results		Proposed Best Results	
		LUT-6	Level count	LUT-6	Level count
arbiter	256/129	403	23	**365**	117
div	128/128	3268	1208	**3267**	1211
hypotenuse	256/128	40385	4527	**40377**	4530
i2c	147/142	224	6	**207**	15
log2	32/32	6570	119	**6567**	119
max	512/130	523	189	**522**	189
mem_ctrl	1204/1231	2117	22	**2086**	23
mult	128/128	4923	90	**4920**	93
priority	128/8	106	26	**103**	26
sin	24/25	1228	55	**1227**	55
sqrt	128/64	3076	1106	**3075**	1106
square	64/128	3244	74	**3242**	76

TABLE 2. Smallest AIG results for the EPFL benchmarks – Comparison with ABC.

Benchmark	I/O	Best-Known Size AIG	Opt. Size AIG
cavlc	10/11	584	483
div	128/128	22206	19250
hypotenuse	256/128	226220	209460
i2c	147/142	769	710
log2	32/32	33213	30522
mem_ctrl	1204/1231	7986	7644
mult	128/128	29885	25371
router	60/30	99	96
sin	24/25	6300	4987
sqrt	128/64	22569	19706
square	64/128	18187	17010
voter	1001/1	13272	9817

show the improvement over 36 industrial benchmarks for ASICs designs.

A. EPFL BENCHMARKS

In this section, we use our global synthesis flow to improve (decrease) the size of the EPFL benchmarks [14]. In particular, we challenge the area (i.e., number of LUTs) category within the EPFL competition,[4] which records the best synthesis results in term of size obtained by mapping the optimized EPFL benchmarks into LUT-6. For these experiments, since we address area optimization, we do not constraint the number of levels during synthesis.

To compare our results, we apply our resynthesis flow followed by the ABC [6] command *"if -K 6 -a"* to map our AIGs into LUT-6. Since LUT-6 minimization does not follow strictly AIG minimization, we adapted our tool to work in general for the LUT-6 experiment in the following way: We inserted selective strashing of LUTs, over previous best results, with optimization and remapping on smaller partitions, to preserve some of the good LUT-6 structures.

Table 1 summarizes our results.[5] At the time of evaluation of this work, we improved 12 of the previous best size (area) results of the EPFL benchmarks,[6] advancing both the size

[4]The best results for the EPFL benchmarks are available at: *https://github.com/lsils/benchmarks*.

[5]Note that we report only those benchmarks for which we had an improvement in area/size.

[6]We compare our results to commit 87cf8ec in *https://github.com/lsils/benchmarks*.

TABLE 3. Detailed results on scalability for *mem_ctrl*.

Flow	Nodes count	Level count	Runtime [s]
Baseline	46836	114	//
ABC resub	46614	114	0.48
Test1 (resub with complex gates)	45993	113	0.4
Test2 (Test1 w/o filtering)	45993	113	**0.8**
ABC_ODC resub	46503	114	5.35
Test3 (resub with FFF)	45852	113	1.5
Test4 (Test3 w/o filtering)	45852	113	**2.3**

results coming from [1] and [43]. Our improvements range from a few LUTs to several (tens) for larger circuits (column **Proposed Best Results** from Table 1). It is worth mentioning that the EPFL benchmarks have been optimized various times in the last 4 years by the most advanced techniques both from industry and academia, thus each improvement (even if relatively small), is highly significant. Note also that some benchmarks (e.g., *max*) are mostly improved by the Boolean difference method, which is capable of resolving convergent logic not distinguished by other techniques. Note that for these experiments, we did not constraint the number of levels (see for instance *i2c* levels are increased from 6 to 15). Furthermore, recently, new best-size results have been presented in [44] and [45]. Even though additional size optimizations have been applied on top of our best results, we still hold the best area results for 3 of the results from Table 1, that is, no further optimizations were found for these cases.

As already pointed out, a smaller AIG was not resulting in the best LUT-6 result for some of the benchmarks. Nevertheless, our global resynthesis flow allows us to obtain the smallest-known AIGs compared to the state-of-the-art, as reported in Table 2. In this scenario, the smallest known AIG from state-of-the-art (**Best-Known Size AIG**) has been computed by strashing the best LUT-6 result and running *resyn2rs* from ABC [6] until no improvement is seen. We present significant improvement; as an example, we show 1.3× (26%) size reduction in the smallest known AIG for the EPFL *voter* benchmark. For some benchmarks, this result is much smaller than the AIG size leading to the best LUT-6 results from Table 1.

The results from Table 1 and 2 show remarkable size optimization on the EPFL benchmarks. To evaluate the scalability of our resubstitution methods and the impact of the proposed filtering techniques, we perform a detailed comparison between our resubstitution (with and without filtering) and ABC on one test case: *mem_ctrl*. All comparisons are performed by reading the (strashed) network into ABC and using truth tables, for the sake of fair comparison. The results are detailed in Table 3. The initial benchmark (**Baseline**) is an AIG with 46836 nodes and 114 levels. By applying the ABC [6] command *"rs -K 10 -N 1"*, the size of the benchmark is reduced to 46614 nodes in 0.48 seconds (**ABC resub** line). On the other hand, our resubstitution method from Alg. 1 (with setting close to ABC, e.g., in cut size, number of divisors, types of nodes, etc.) obtains 45993 nodes in

TABLE 4. Post place&route results on 36 industrial design for ASICs.

Flow	Comb. Area	N-Comb. Area	WNS	TNS	Comb. SW Pow.	Comb. Leak Pow.	# Cells	Runtime
Baseline	-	-	-	-	-	-	-	-
Ex1: optimized AIG	-0.48%	+0.02%	+0.02%	+1.09%	-0.68%	-0.22%	-0.45%	+0.2%
Ex2: Ex1 + novel resub	-1.31%	+0.04%	-0.65%	-0.24%	-1.29%	-0.45%	-0.95%	+0.5%
Ex3: Ex2 + Bdiff resub	-1.87%	+0.01%	+0.11%	+0.87%	-1.36%	-0.36%	-1.24%	+1.2%
Ex4: Ex3 + het. kernel	-2.47%	-0.01%	-0.80%	-2.43%	-2.18%	-0.90%	-2.07%	+1.8%
Ex5: Ex4 + SAT resub	**-3.12%**	**-0.02%**	**-1.34%**	**-0.82%**	**-2.49%**	**-0.98%**	**-2.25%**	+2.2%

"Comb. Area" and "N-Comb. Area" are the combinational and non-combintional area respectively. "WNS" is the worst negative slack, and "TNS" is the total negative slack; "Comb. SW Pow." is the dynamic power dissipation due to switching activity, while "Comb. Leak Pow. " is the static power dissipation due to leakage. " # Cells" is the total cell count in the design, and the "Runtime" is the total runtime post P&R. All results are averaged over 36 designs.

0.4 seconds (compare to **Test1** from Table 3). **Test2** presents results of resubstitution from Test1, without the filtering technique for 1-resubstitution. The experiments show that the size is optimized of the same amount, but the runtime is 2×. The second part of Table 3 shows results when resubstitutions are improved with MSPF (i.e., observability don't care) computations. In particular, **ABC_ODC resub** shows results obtained by running the ABC command "*rs -K 10 -N 1 -v -F 4*" on the baseline benchmark. This case obtains better QoR as compared to **ABC resub**, but the runtime increases significantly. The results for our resubstitution with FFF (Alg. 5) are shown as **Test3** and **Test4**. In particular, results with and without filtering techniques are shown, respectively. As compared to ABC, our method shows improved QoR and smaller runtime. Moreover, the runtime obtained without the filtering rules (**Test4**) is increased, while the node and level count remain the same. This confirms the importance of our proposed filtering technique to attempt higher quality resubstitution in the context of logic optimization.

To further stress the effect of filtering techniques, consider a big industrial benchmark having 113533 PIs and 17305 POs, respectively, and 1271692 AIG nodes and 267 levels. The ABC command *"rs -K 10 -N 1"* reduces the size (levels) to 1270497 (262), in 14.48 seconds. Our resubstitution method from Alg. 1 results instead in size (levels) of 1269219 (264) in only 6.0 seconds. When filtering techniques are turned off, the QoR is not affected, but the runtime increases to 9.9 seconds.

B. ASIC RESULTS

We tested a commercial EDA flow, empowered with our global resynthesis flow, on 36 state-of-the-art ASICs, coming from major electronics industries. Due to non-disclosure agreements, we cannot provide details on each ASIC benchmark. However, we present average results w.r.t. a baseline flow without our resynthesis methods. The post place & route results are summarized in Table 4. All benchmarks are verified with an industrial formal equivalence checking flow. In Table 4, the baseline is a complete industrial EDA flow from register transfer level to GDSII, without any of the presented techniques. The first experiment presents results when our gradient-based AIG minimization is included in the EDA flow. On average, this technique decreases the combinational area, which is our target metric, of 0.48%, which negligible increase in runtime and WNS/TNS. The second experiment enriches experiment 1 with our novel resubstitution methods. These techniques allow a further combinational area decrease, resulting in improvements also for WNS and TNS. Experiment 3 uses the resubstitution with Boolean difference to further improve the combinational area, while experiment 4 enhances the previous steps by using heterogeneous elimination for kernel extraction. The last experiment presents results when the complete flow from Section VI is applied, i.e., experiment 4 enriched with SAT resubstitution.

Our complete design flow embedding our new optimization, and highlighted in green, enables sensible combinational area and dynamic power (without considering the clock network) reductions, 3.12% and 2.49%, respectively. On average, we also achieve WNS/TNS improvements, with a runtime increase of only 2.2%.

VIII. CONCLUSION

The continuous push to advance QoR in the EDA community has resulted in a revived interest for Boolean methods. Boolean methods (e.g., Boolean resubstitution) are universally considered runtime-expensive, and thus are used cautiously in logic synthesis flows. In this paper, we presented three novel Boolean resubstitution methods, more scalable and runtime-effective as compared to state-of-the-art. The former method uses Boolean filtering techniques and windows to speed up the candidates selection; the second method exploits a weaker notion of MSPF to accelerate the don't cares computation; the latter method presents a fast resubstitution method based on the Boolean difference computation over network partitions. While the three resubstitution algorithms are independent of the reasoning engine used for their implementation, the choice of such engine can determine their scalability and QoR. We thus presented practical scenarios, depending on circuits characteristics and optimization opportunities, in which our methods are best driven by truth tables, binary decision diagrams, satisfiability, or a blend of those.

The three Boolean resubstitution methods have been implemented together with well-known and optimized state-of-the-art algorithms to build a global resynthesis flow that addresses size optimization. Altogether, our global resynthesis flow achieves substantial optimization results. We have obtained significant improvements over the smallest known AIGs for EPFL benchmarks, and have improved best-known area results in the EPFL synthesis competition. We have

demonstrated the efficacy of our Boolean resynthesis flow over 36 industrial designs for ASICs, which resulted in 3.12% combinational area savings and 2.49% dynamic power reduction, after physical implementation. The proposed flow guarantees excellent scalability, achieving, on average, a limited increase in the runtime of only 2.2%.

As part of future work, we envision more filtering and optimizations can be found for specific data-structures. For this purpose, the implementation of the presented algorithms within state-of-the-art open source tools, as the EPFL Libraries or ABC, will be part of our future research. The implementation of our resubstitution algorithms using data-structures ranging from AIG to MIGs or XAGs will be an interesting direction of logic synthesis research and give a useful comparison point for the proposed industrial algorithms.

REFERENCES

[1] L. Amaru, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. D. Micheli, "Improvements to Boolean resynthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 755–760.

[2] E. Testa, L. Amaru, M. Soeken, A. Mishchenko, P. Vuillod, J. Luo, C. Casares, P.-E. Gaillardon, and G. D. Micheli, "Scalable Boolean methods in a modern synthesis flow," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1643–1648.

[3] H. Riener, E. Testa, L. Amaru, M. Soeken, and G. D. Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. 14th IEEE/ACM Int. Symp. Nanosc. Archit.*, Jul. 2018, pp. 157–162.

[4] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, no. 2, pp. 264–300, Feb. 1990.

[5] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994.

[6] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. Int. Conf. Comput.-Aided Verification*, 2010, pp. 24–40.

[7] M. Damiani, J. C.-Y. Yang, and G. D. Micheli, "Optimization of combinational logic circuits based on compatible gates," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 11, pp. 1316–1327, Nov. 1995.

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

[9] S. Muroga, "Logic synthesizers, the transduction method and its extension, Sylon," in *Logic Synthesis and Optimization*. New York, NY, USA: Springer, 1993, pp. 59–86.

[10] L. Amaru, P. Vuillod, J. Luo, and J. Olson, "Logic optimization and synthesis: Trends and directions in industry," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1303–1305.

[11] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Reading, MA, USA: Addison-Wesley, 2015.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.

[14] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. 24th Int. Workshop Log. Synth.*, 2015, pp. 1–5.

[15] R. K. Brayton and C. T. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. Int. Symp. Circuits Syst.*, 1982, pp. 49–54.

[16] E. Testa, M. Soeken, L. G. Amar, and G. De Micheli, "Logic synthesis for established and emerging computing," *Proc. IEEE*, vol. 107, no. 1, pp. 165–184, Jan. 2019.

[17] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. New York, NY, USA: Springer, 2006.

[18] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 7, no. 6, pp. 723–740, Jun. 1988.

[19] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 5, pp. 743–755, May 2006.

[20] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 597–604.

[21] R. Drechsler and B. Becker, *Binary Decision Diagrams: Theory and Implementation*. New York, NY, USA: Springer, 2013.

[22] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Reading, MA, USA: Addison-Wesley, 2011.

[23] R. K. Brayton and F. Somenzi, "An exact minimizer for Boolean relations," in *Proc. IEEE Int. Conf. Computer-Aided Design. Dig. Tech. Papers*, Dec. 1989, pp. 316–319.

[24] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 7, pp. 866–876, Jul. 2002.

[25] L. Amaru, P.-E. Gaillardon, and G. D. Micheli, "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*, 2013, pp. 47:1–47:6.

[26] A. Mishchenko, R. K. Brayton, J. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, p. 34:1–34:23, 2011.

[27] A. Mishchenko, R. K. Brayton, A. Petkovska, and M. Soeken, "SAT-based optimization with dont-cares revisited," in *Proc. Int. Workshop Log. Synth.*, 2017.

[28] K. Debnath, R. Murgai, M. Jain, and J. Olson, "SAT-based redundancy removal," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 315–318.

[29] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1649–1654.

[30] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 830–835.

[31] S. P. Khatri and K. Gulati, Eds., *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. New York, NY, USA: Springer, 2011.

[32] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. 43rd Annu. Conf. Design Autom. (DAC)*, 2006, pp. 532–535.

[33] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. Int. Workshop Log. Synth.*, 2006, pp. 15–22.

[34] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita, "Boolean resubstitution with permissible functions and binary decision diagrams," in *Proc. Conf. Proc. 27th ACM/IEEE Design Autom. Conf. (DAC)*, 1990, pp. 284–289.

[35] A. M. Y. Miyasaka and M. Fujita, "A simple BDD package without variable reordering and its application to logic optimization with permissible functions," in *Proc. Int. Workshop Log. Synth.*, 2019, pp. 1–8.

[36] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization," in *Proc. 41st Annu. Conf. Design Autom. (DAC)*, 2004, pp. 438–441.

[37] S. Kauffman, "The large scale structure and dynamics of gene control circuits: An ensemble approach," *J. Theor. Biol.*, vol. 44, no. 1, pp. 167–190, Mar. 1974.

[38] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.

[39] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 205–213, Feb. 1991.

[40] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," 1991.

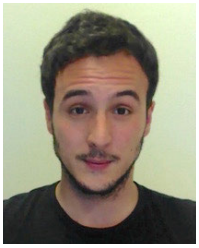
[41] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. 55th Annu. Design Autom. Conf.*, Jun. 2018, p. 50.

[42] H. S. Warren, *Hacker's Delight*. Reading, MA, USA: Addison-Wesley, 2002.

[43] L. Machado and J. Cortadella, "Support-reducing functional decomposition for FPGA technology mapping," in *Proc. Int. Workshop Log. Synth.*, 2018, pp. 1–8.

[44] L. Machado and J. Cortadella, "Support-reducing decomposition for FPGA mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 1, pp. 213–224, Jan. 2020.
[45] I. Lemberski, A. Suponenkovs, and M. Uhanova, "LUT-oriented asynchronous logic design based on resubstitution," in *Proc. 14th Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Apr. 2019, pp. 1–4.

ELEONORA TESTA (Member, IEEE) received the Ph.D. degree in computer science from the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland, in 2020. She is currently a Research and Development Engineer with the Design Group, Synopsys Inc., Zurich, Switzerland. Her research interests include logic synthesis, electronic design automation, and post-CMOS technologies.



LUCA AMARÚ (Member, IEEE) received the Ph.D. degree in computer science from the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland, in 2015. He is currently a Senior Research and Development Manager with the Design Group, Synopsys Inc., Sunnyvale, CA, USA, where he is responsible for designing efficient data structures and algorithms for logic synthesis. Prior to joining Synopsys, he was a Visiting Researcher with Stanford University and a Research Assistant with EPFL. His current research interests include electronic design automation, logic in computer science, and beyond CMOS technologies. He has been serving as a TPC member for several conferences, including DATE, IWLS and DSD. He is reviewer for several IEEE journals. He was a recipient of the IEEE TCAD Donald O. Pederson Best Paper Award in 2018, the EDAA Outstanding Dissertation Award in 2015, the Best Presentation Award at FETCH Conference in 2013, and a Best Paper Award Nomination at ASP-DAC Conference also in 2013. He received fellowships and research contribution awards from EPFL.

MATHIAS SOEKEN (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2013. He is currently a Scientist with the École Polytechnique Fédéderale de Lausanne (EPFL), Lausanne, Switzerland. He is investigating constraint-based techniques in logic synthesis and industrial-strength design automation for quantum computing. He is actively maintaining the logic synthesis frameworks CirKit and RevKit. His current research interests include the many aspects of logic synthesis and formal verification. He has been serving as a TPC member for several conferences, including DAC, DATE, and ICCAD, and is a Reviewer for *Mathematical Reviews* as well as for several other journals. He was a recipient of the scholarship from the German Academic Scholarship Foundation.

ALAN MISHCHENKO (Senior Member, IEEE) 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 Electrical Engineering and Computer Science Department (EECS), University of California, Berkeley, where he is currently a Full Researcher. His research interests include computationally efficient logic synthesis and formal verification.

PATRICK VUILLOD received the Ph.D. degree from INP Grenoble with an exchange from Stanford University. He is currently a part of the Synopsys Research and Development Center, Grenoble, France. He has been working on Synopsys Implementation tools for 20 years. He has been developing optimization flows of Design Compiler, ICC2 compiler, and the recently announced Fusion Compiler. He worked on a broad area of topics, such as logic optimization, multi-threaded delay optimization, routing congestion estimation algorithms, and physical optimization, allowing several patents and articles.

PIERRE-EMMANUEL GAILLARDON (Senior Member, IEEE) received the Electrical Engineer degree from CPE-Lyon, France, the M.Sc. degree in electrical engineering from INSA Lyon, France, the Ph.D. degree in electrical engineering from CEA-LETI, Grenoble, France, and the Ph.D. degree in electrical engineering from the University of Lyon, France. He is currently an Associate Professor and the Associate Chair for academics and strategic initiatives of the Electrical and Computer Engineering (ECE) Department, The University of Utah, Salt Lake City, UT, USA. Previously, he was a Research Associate with the Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland. His research interests include the development of novel computing systems exploiting emerging device technologies and novel EDA techniques. He was a recipient of the 2017 NSF CAREER Award, the 2018 IEEE CEDA Pederson Award, the 2019 DARPA Young Faculty Award, and the 2019 IEEE CEDA Ernest S. Kuh Early Career Award.

GIOVANNI DE MICHELI is currently a Professor of electrical engineering and a Professor of computer science with EPF Lausanne, Switzerland. His research interests include several aspects of design technologies for integrated circuits and systems, such as design and synthesis for emerging technologies. He is also a Fellow of ACM, a member of the Academia Europaea, and an International Honorary member of the American Academy of Arts and Sciences. He was a recipient of the 2019 ACM/SIGDA Pioneer Award, the 2016 IEEE/CS Harry Goode Award for seminal contributions to design and design tools of Networks on Chips, the 2016 EDAA Lifetime Achievement Award, the 2012 IEEE/CAS Mac Van Valkenburg award for contributions to theory, practice and experimentation in design methods and tools, and the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems.