Heuristic Logic Resynthesis Algorithms at the Core of Peephole Optimization

Siang-Yun Lee and Giovanni De Micheli, *Life Fellow, IEEE*

***Abstract*—Logic resynthesis is one of the core problems in modern peephole logic optimization algorithms. Given a target function and a set of existing functions, logic resynthesis asks for a circuit reusing some of the existing functions and generating the target. While exact methods, such as enumeration and SAT-based synthesis, guarantee optimal solutions, limitations on the problem size are inevitable due to scalability concerns. In this work, we propose heuristic resynthesis algorithms for AND-based, majority-based, and multiplexer-based circuits, which are scalable in all aspects. Used as the core of high-effort optimization, our heuristic resynthesis algorithms play a key role in enabling 2%–3% further size reduction on benchmarks that are already processed by state-of-the-art optimization flows.**

***Index Terms*—Boolean resubstitution, combinational circuit, logic synthesis, peephole optimization, resynthesis.**

I. Introduction

LOGIC synthesis plays an important role in modern electronic design automation flows, optimizing gate-level netlists, and removing redundant logic in them [1], [2], [3]. *Peephole optimization* is a divide-and-conquer strategy to maintain scalability of logic synthesis algorithms, where small portions of a circuit, often referred to as *windows* or *cuts*, are extracted, optimized independently, and substituted back. With the large scale of designs nowadays, most logic synthesis algorithms, such as rewriting [4], [5], [6], [7], resubstitution [4], [8], [9], [10], refactoring [4], [9], [11], etc., fall into the category of peephole optimizations.

One of the important steps in any peephole optimization algorithm is resynthesizing the extracted subcircuit into a better one. In this work, we define the *logic resynthesis* problem as a generalized formulation of this step: the problem is given a *target* function, which is usually the root of a cut or the output(s) of a window, and some *divisor* functions, which are existing functions from neighboring nodes in the network. The resynthesis problem asks for a *dependency circuit*, computing a *dependency function*, that takes as inputs a subset of divisor functions and generates the target function at the output. If the solution is better than the original subnetwork in the predefined cost metric, then it can be used to substitute the targeted node.

Various resynthesis strategies are adopted by different logic synthesis algorithms. For example, in cut rewriting, the divisor functions are always the projection (identity) functions and the target function has a small number of inputs (usually 4), thus the optimal dependency circuit can be looked up from a precomputed database [4], [5] or be synthesized by SAT solving [6], [7]. As another example, in refactoring, the divisor functions are also the projection functions, but the dependency circuit is synthesized by two-level logic optimization [4], [9]. In contrast, in resubstitution, divisor functions other than only the projection functions are collected and used as stepping stones to construct the target function. As the number of all possible sets of divisor functions is very large, a resubstitution algorithm has to investigate the divisor functions and resynthesize the dependency circuit on the fly. Previous resubstitution works mostly attempt to enumerate small dependency circuits and compare them to the target function [4], [9], [10]. The drawback of this approach is that the dependency circuit is limited to a small size, as otherwise the search space becomes too big.

With the introduction of the simulation-guided logic synthesis paradigm [12], where simulation signatures are used to approximate global logic functions, it becomes affordable to extend the window sizes in peephole optimization. Craving for better optimization effort, resynthesis methods capable of optimizing more complex functions, which require larger dependency circuits, are in need. In a highly optimized network where rewriting with a small cut size cannot make any further optimization, there may still be hidden optimization opportunities requiring the involvement of a larger portion of the network. In some cases, not only a larger cut (and thus a larger window) needs to be considered but the resynthesized subnetworks should also not be limited to small ones.

In this article, we research on the problem of logic resynthesis and propose resynthesis algorithms for networks based on AND, MAJ, or MUX gates, targeting size optimization.[1] The proposed algorithms share the following characteristics.

1) *Support for Incomplete Functions and Don't Cares:* The divisor and target functions may be given as completely specified Boolean functions or partial simulation signatures [12]. The algorithms resynthesize dependency

Manuscript received 7 November 2022; revised 24 January 2023; accepted 3 March 2023. Date of publication 13 March 2023; date of current version 20 October 2023. This work was supported by the SNF Grant "Supercool: Design Methods and Tools for Superconducting Electronics," under Grant 200021_1920981. This article was recommended by Associate Editor L. Amaru. *(Corresponding author: Siang-Yun Lee.)*

The authors are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland (e-mail: siang-yun.lee@epfl.ch).

Digital Object Identifier 10.1109/TCAD.2023.3256341

[1]This manuscript is an extension to and summary of two of the authors' previous works: AND-based resynthesis was first proposed in [13] and MAJ-based resynthesis was first proposed in [14], whereas MUX-based resynthesis is new in this manuscript.

circuits satisfying the given parts of functions and make no assumption on the uninformed parts. Moreover, don't cares of the target function may be given, and the algorithms take advantage of this information to resynthesize smaller dependency circuits.

2) *Heuristic But Unlimited:* Optimality may only be guaranteed when the optimal solution is small. It is also not guaranteed that a solution is always found. Nevertheless, there is no limit on the possible solution size. When a small-sized solution does not exist, the heuristic may still find a bigger solution which exact methods can never find within reasonable runtime.
3) *Top-Down Decomposition:* Although the three proposed algorithms are designed differently, they all start from choosing "good" divisors based on some evaluation criteria involving the target function. Then, if the target cannot be realized within a few gates, it is decomposed into easier-to-realize targets by a gate on top.

The proposed heuristic resynthesis algorithms have better complexities comparing to existing exact algorithms, while compromising with little overhead in the quality of result comparing to optimal solutions. With their high efficiency and unlimited problem size, heuristic resynthesis is the only practical candidate to serve as the core of high-effort peephole optimization. Experimental results show that our proposed techniques enable 2%–3% additional size reduction on benchmarks which are already highly optimized by state-of-the-art flows, achieved within less than 50% runtime of the state-of-the-art flows.

II. Preliminaries and Problem Formulation

A. Boolean Functions and Truth Tables

A *Boolean variable* is a variable taking values in the *Boolean domain* $\mathbb{B} = \{0, 1\}$, and a *Boolean function* is a function of Boolean variables. Unless otherwise specified, all functions in the remaining of this article are single-output Boolean functions.

There are several possible representations of Boolean functions, such as propositional formulas, Boolean chains [15], binary decision diagrams [16], and truth tables. We use the conventional Boolean operators when writing propositional formulas ($\neg$ for NOT, $\wedge$ for AND, $\vee$ for OR, $\oplus$ for XOR, and $\leftrightarrow$ for XNOR). In this article, the *truth table* $T[f]$ of a k-input Boolean function $f : \mathbb{B}^k \to \mathbb{B}$ is a bit-string $u = u_1 \cdots u_l$, i.e., a sequence of bits, of *length* $l = 2^k$. The bit $u_i \in \mathbb{B}$ at the ith position ($0 \le i < l$), denoted as $T[f]_i$, is equal to the output of f under the input assignment $\vec{a} = (a_1, \ldots, a_k)$, where

$$2^{k-1} \cdot a_k + \cdots + 2^0 \cdot a_1 = i. \tag{1}$$

The assignment $\vec{a} \in \mathbb{B}^k$ is also called a *minterm* in the input space of f. If $T[f]_i = f(\vec{a}) = 1$, $\vec{a}$ is said to be an *onset* minterm; otherwise, if $T[f]_i = f(\vec{a}) = 0$, $\vec{a}$ is said to be an *offset* minterm.

We use

$$\textsc{Ones}(f) = \sum_{i=0}^{l-1} T[f]_i \tag{2}$$

to denote the number of 1 bits in the truth table of f, which is also the number of onset minterms, or the size of the onset.

Truth tables are manipulated by carrying out the usual Boolean operations on all of their bits. Suppose that $u = u_1 \cdots u_l$ and $v = v_1 \cdots v_l$ are two truth tables of length l, and $\alpha : \mathbb{B} \to \mathbb{B}$ and $\beta : \mathbb{B}^2 \to \mathbb{B}$ are, respectively, unary and binary Boolean operations, then $\alpha(u) = \alpha(u_1) \cdots \alpha(u_l)$ and $\beta(u, v) = \beta(u_1, v_1) \cdots \beta(u_l, v_l)$. Such truth table manipulations can be highly efficiently implemented with the bit-parallel operations supported by modern CPUs [17]. The bits of the truth tables are split into buckets of 32- or 64-bit machine words and each bucket is processed in one machine instruction.

B. Logic Resynthesis

Logic resynthesis (or simply *resynthesis*) is the problem of re-expressing a function in terms of other functions.

Problem Formulation 1 (Resynthesis): Given a *target function* (or simply *target*) $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = (x_1, \ldots, x_k)$ and a collection $G = \{g_1, \ldots, g_n\}$ of n *divisor functions* (or simply *divisors*) $g_i : \mathbb{B}^k \to \mathbb{B}$, $1 \le i \le n$ over the same variables, find a *dependency function* $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})) \ \forall \vec{x} \in \mathbb{B}^k. \tag{3}$$

In this formulation, variables $x_1, \ldots, x_k$ are not inputs of the function h, but any subset of them may be embedded as divisors by defining, for example, $g_1(\vec{x}) = x_1$. Also, the expression of h does not necessarily depend on all of its n inputs. In practice, a resynthesis problem may be further restricted by, for example, a set of logic operations or the number of operations allowed to be used in the expression of the dependency function. This will be further introduced in Section II-D.

Example 1 (Unrestricted Resynthesis): Given the target function

$$f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (\neg x_2 \wedge \neg x_3) \tag{4}$$

and the divisor set

$$G = \begin{Bmatrix} g_1(x_1, x_2, x_3) = x_1 \wedge \neg x_2 \\ g_2(x_1, x_2, x_3) = \neg x_2 \wedge x_3 \\ g_3(x_1, x_2, x_3) = x_3 \\ g_4(x_1, x_2, x_3) = x_1 \leftrightarrow x_2 \end{Bmatrix} \tag{5}$$

one possible dependency function is

$$h(g_1, g_2, g_3, g_4) = (g_1 \vee g_4) \wedge \neg g_2. \tag{6}$$

Notice that (3) is satisfied because

$$\begin{aligned} h &= ((x_1 \wedge \neg x_2) \vee (x_1 \leftrightarrow x_2)) \wedge \neg(\neg x_2 \wedge x_3) \\ &= (x_1 \wedge x_2) \vee (\neg x_2 \wedge \neg x_3) = f. \end{aligned} \tag{7}$$

The resynthesis problem can be seen as a generalization of the classical *logic synthesis* problem, where an expression or realization of h over the same variables $x_1, \ldots, x_k$ as f is sought for, i.e., G is restricted to $\{g_1 = x_1, \ldots, g_k = x_k\}$. Logic resynthesis is different from *logic decomposition* [18], [19] or *functional decomposition* [20], [21], where the problem is not limited to a given divisor collection G, but involves

identifying the needed divisors. In contrast, solving resynthesis problems can be seen as the core step in a *resubstitution* algorithm [4], [8], [9], [10].

C. Logic Networks

Logic networks, or simply *networks*, are gate-level representations of digital circuits commonly used as the data structure during logic optimization. Networks are directed acyclic graphs (DAGs), where nodes model logic gates chosen from a predefined set, and edges model interconnecting wires. Edges may optionally be tagged as being *complemented*, representing an inverter on the wire. Incoming edges of a node are called *fanins*, whereas outgoing edges are called *fanouts*. For convenience, nodes having fanouts pointing to a common node (i.e., fanin nodes of a node) are said to be *siblings* of each other. The size of a network is determined by its number of nodes, whereas inverters are, in this article, not counted toward the network size.

Prominent examples of logic networks include *and-inverter graphs* (AIGs), where each node represents a two-input AND gate, and majority-inverter graphs (MIGs) [22], where each node represents a three-input majority (MAJ) gate. The MAJ gate computes the majority function M of its fanins [23], i.e.,

$$M(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3). \tag{8}$$

Extending the gate library with XOR gates, the *Xor-AIG* (XAG) is a logic network where nodes can be either a two-input AND gate or a two-input XOR gate. Another interesting type of networks is the *multiplexer-inverter graph* (MuxIG), where each node represents a 2-to-1 *multiplexer* (MUX) gate. The MUX gate has three nonsymmetric inputs: 1) the S-input as the selection ("if") signal; 2) the T-input as the "then" signal; and 3) the E-input as the "else" signal. The function computed by a MUX gate can be written as

$$\text{MUX}(s, t, e) = (s \wedge t) \vee (\neg s \wedge e). \tag{9}$$

D. Peephole Optimization

Peephole optimization is a class of logic optimization algorithms that repeatedly select a small subnetwork, optimize it, and replace the subnetwork with an optimized one. A number of well-known logic optimization algorithms fall into this category, such as cut rewriting [5], [7], resubstitution [8], [9], [12], and window rewriting [13]. Logic resynthesis can be used in the second step of peephole optimization, i.e., optimizing the selected subnetwork by resynthesizing the output function(s) of the subnetwork. For details in the other steps, such as cut computation, windowing, collecting divisors, and evaluation of candidate replacement subnetworks, we refer the interested readers to the literature cited above. In this work, we focus on the resynthesis problem for AND-based, MAJ-based, and MUX-based circuits targeting size optimization. That is, the dependency function h is represented by an AIG, XAG, MIG, or MuxIG, called the *dependency circuit*, and the optimization goal is minimizing its size.

Example 2 (MIG Resynthesis Targeting Size Optimization): Given the target function

$$f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3 \tag{10}$$

and the divisor set

$$G = \Big\{ g_1(x_1, x_2, x_3) = x_1 \\ g_2(x_1, x_2, x_3) = x_2 \\ g_3(x_1, x_2, x_3) = x_3 \\ g_4(x_1, x_2, x_3) = M(\neg x_1, x_2, x_3) \\ g_5(x_1, x_2, x_3) = M(\neg x_1, \neg x_2, x_3) \Big\} \tag{11}$$

extracted from an MIG by a peephole optimization algorithm. The resynthesis problem is restricted to use only majority gates and inverters, and solutions with fewer gates are preferred. One possible dependency function is

$$h(g_1, g_2, g_3, g_4) = M(\neg g_2, g_4, \neg g_5) \tag{12}$$

whose corresponding dependency circuit has the least possible size of 1.

E. Don't-Care-Based Optimization

Most modern logic optimization algorithms place emphasis on the computation and utilization of *don't cares*, which are flexibilities in logic functions [24]. The peephole optimization algorithms mentioned in Section II-D are all examples of don't-care-based optimization [5], [7], [8], [9], [12], [13]. When solving the resynthesis problem as part of peephole optimization, it is important to take the computed don't cares into account. Although don't cares may come from different sources, namely, satisfiability don't cares and observability don't cares, they can be treated the same when formulating the resynthesis problem. Formally, the *don't-care set* of a single-output Boolean function is defined as the set of minterms (i.e., input value assignments) for which the output value is allowed to be either 0 or 1.

Problem Formulation 2 (Resynthesis With Don't Cares): Given a target function $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = (x_1, \ldots, x_k)$, a *don't-care set* $D \subseteq \mathbb{B}^k$, and a collection $G = \{g_1, \ldots, g_n\}$ of n divisor functions $g_i : \mathbb{B}^k \to \mathbb{B}, 1 \leq i \leq n$ over the same variables, find a dependency function $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})) \ \forall \vec{x} \in \mathbb{B}^k \backslash D. \tag{13}$$

For convenience, we define the *care set* $C = \mathbb{B}^k \backslash D$ and the *care function* $c : \mathbb{B}^k \to \mathbb{B}$, where

$$c(\vec{x}) = \begin{cases} 1, & \vec{x} \in C \\ 0, & \vec{x} \in D. \end{cases} \tag{14}$$

Thus, (13) is equivalent to

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})) \ \forall \vec{x} \in \mathbb{B}^k \text{ s.t. } c(\vec{x}) = 1. \tag{15}$$

Moreover, if we define the target *onset function* $f_{\text{on}} = f \wedge c$ and the *offset function* $f_{\text{off}} = \neg f \wedge c$, then (13) is also equivalent to

$$h\Big(g_1(\vec{x}), \ldots, g_n(\vec{x})\Big) \implies \neg f_{\text{off}}(\vec{x}) \text{ and} \\ f_{\text{on}}(\vec{x}) \implies h\Big(g_1(\vec{x}), \ldots, g_n(\vec{x})\Big) \forall \vec{x} \in \mathbb{B}^k. \tag{16}$$

Example 3 (Resynthesis With Nonempty Don't-Care Set): Suppose we have the same target function f and divisor set G

as in Example 1 [(4) and (5), respectively]. Additionally, we are now given the care function

$$c(x_1, x_2, x_3) = x_2 \vee (x_1 \leftrightarrow x_3).$$

In other words, the don't-care set $D = \{(1, 0, 0), (0, 0, 1)\}$ is nonempty. For this relaxed problem, one possible dependency function is

$$h(g_1, g_2, g_3, g_4) = g_4 \tag{17}$$

which is simpler than (6) thanks to the provided don't cares. Notice that (13) is satisfied because the difference between f and h ($f \oplus h = \{(1, 0, 0), (0, 0, 1)\}$) does not intersect with the care set.

F. Simulation-Guided Logic Synthesis

The *simulation-guided paradigm* [12] is a logic synthesis and verification model where partial simulation signatures are used to approximate the global functions of nodes in a network. In this paradigm, a nonexhaustive set of simulation patterns (i.e., value assignments to primary inputs) is generated and used to simulate the network. The simulated values, called *simulation signatures*, at each node in the network are approximations of their global function and can be used to resynthesize dependency circuits.

The resynthesis algorithms proposed in this article are compatible with the simulation-guided paradigm. In this case, the target and divisor functions are represented by the simulation signatures of the corresponding nodes in the network and *partial truth tables* are used as the data structure. A partial truth table is a truth table of arbitrary length l, representing a partially specified, incomplete function $f : X \to \mathbb{B}$, where $X \subseteq \mathbb{B}^k$ and k is the number of primary inputs of the network. The ith bit $T[f]_i$ is the output of f under the ith simulation pattern in the set. What the pattern actually is not important for the resynthesis problem. It is only required that the partial truth tables of the target and divisors use the same ordered set of simulation patterns.

Problem Formulation 3 (Resynthesis With Incompletely Specified Functions): Given a target function $f : X \to \mathbb{B}$ and a collection $G = \{g_1, \ldots, g_n\}$ of n divisor functions $g_i : X \to \mathbb{B}, 1 \leq i \leq n$ defined over the same input space $X \subseteq \mathbb{B}^k, k \in \mathbb{N}^+$, find a dependency function $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})) \ \forall \vec{x} \in X. \tag{18}$$

Optionally and similarly to the problem formulation in Section II-E, a don't-care set $D \subseteq X$ may be given. The care set is then $C = X \backslash D$, and the care function $c : X \to \mathbb{B}$ is defined the same as in (22).

A resynthesis algorithm receiving target and divisor functions as truth tables does not distinguish the case where functions are incompletely specified from where they are completely specified. A solution given by the algorithm fulfills (18), and it is up to the simulation-guided framework to validate the dependency circuit in the context of the network and add more bits into the partial truth tables to block invalid solutions [12].

III. Related Works

In this section, we introduce previous works dealing with the same or similar problems.

A. Functional Dependency by Interpolation

In [25], a method to find *functional dependency* using interpolation was proposed. The problem of finding functional dependency is essentially the same as the unrestricted logic resynthesis problem (Problem Formulation 1), where the goal is only to find a dependency function without a particular focus on (minimizing) the corresponding dependency circuit. In [25], given a target function f and a set of base functions G (i.e., divisor functions in our terminology), it is first checked if f functionally depends on G, i.e., if a dependency function h exists. This is done by solving a *satisfiability* (SAT) problem consisting of two copies of the circuit representation of f and G and additional constraints that the outputs of G are the same, but one copy outputs $f = 0$ and the other outputs $f = 1$. Intuitively, the SAT problem encodes that there exists a pair of offset $\vec{x}_0$ and onset $\vec{x}_1$ minterms of f, such that $g_i(\vec{x}_0) = g_i(\vec{x}_1)$ for all $g_i \in G$. A dependency function h exists if and only if the SAT instance is unsatisfiable, and such h can be computed by deriving the interpolant from the refutation proof given by the SAT solver.

The interpolation-based method was later used in [8] as part of resubstitution for look up table (LUT) networks. Because the dependency function is implemented as an LUT node, it is not needed to construct a dependency circuit. However, for resubstitution algorithms for AIGs, XAGs, or MIGs, etc., the size of the dependency circuit is crucial for the optimization quality. Thus, the interpolation-based method is not applicable there. Also, as the procedure involves constructing conjunctive normal form (CNF) clauses of a circuit computing f and G, it cannot solve the resynthesis problem with incomplete simulation signatures (Problem Formulation 3).

B. SAT-Based Exact Synthesis

SAT solving can also be used to find the *smallest* dependency circuit, instead of just *some* feasible dependency function. SAT-based exact synthesis of Boolean chains encodes the following question into a CNF formula: "Does there exist a Boolean chain which implements the given function f with exactly r steps[2]?" A solution Boolean chain can be interpreted from a satisfiable assignment to the encoded CNF formula, whereas an unsatisfiable result means a solution of r steps is impossible. By solving such SAT problem iteratively with different values of r, the smallest feasible r can be found [15]. While SAT-based exact synthesis was originally described to synthesize a Boolean chain computing a given function at its output(s) in terms of its input variables, i.e., it solves a subset of the resynthesis problem where divisors are projection functions, it can be modified and extended to solve the general resynthesis problem where divisors can be any

[2]Using the terminology in this article, a Boolean chain with r steps is a logic network with r nodes, where each node models an arbitrary logic gate. Additional clauses may be added to the CNF formula to constrain possible gate types to a predefined set.

functions and don't cares are supported [7]. In [26], different CNF encodings of the problem were analyzed and compared. However, although it is possible to reduce the number of variables involved in the SAT instance, it is done at the cost of more clauses in the CNF formula. As the intrinsic complexity of the problem is exponential, the scalability of an exact algorithm is always limited.

C. Enumeration-Based Resubstitution

Resubstitution is a logic optimization technique which substitutes a node in the network with another existing node, or with newly created nodes constructed upon other existing nodes [3]. Resubstitution for AIG size minimization was first proposed in [4], where windows of no more than 16 inputs are constructed to collect structurally proximate divisor nodes and to perform complete local simulation. Small subnetworks of up to three AND gates and taking divisors as inputs are enumerated, simulated, and compared to the target function. If the composed function is the same as (or compatible subject to the care set) the target, a viable dependency circuit is found. Such search for resubstitutions is essentially the AIG resynthesis problem with size awareness. The complexity of the enumeration-based resynthesis approach is $\mathcal{O}(|G|^{|H|+1})$, where $|G|$ is the number of divisors and $|H|$ is the size of possible dependency circuit. Thus, $|G|$ is limited to at most 150 and $|H|$ is limited to at most 2 in [4].

In [9], enumeration-based resynthesis was extended to larger dependency circuits, but still limited to some predefined structures, such as AND-XOR, MUX, MUX-XOR, etc. A Boolean filtering rule was proposed to filter out useless divisors, so that the search space was reduced. Overall, eight types of dependency circuit structures are tried in the increasing order of their size, and for each structure, filtered set of divisors are enumerated at the inputs similarly to [4].

An enumeration-based resubstitution for MIGs was first proposed in [10]. The algorithm enumerates dependency circuits of up to two MAJ gates. Two efficiency enhancement techniques were proposed.

1) A filtering rule derived from the majority law is applied

$$\text{if } x \neq y \text{ and } \exists z, M(x, y, z) = f, \text{ then } M(x, y, f) = f. \quad (19)$$

2) As a preprocessing step, the truth tables are normalized to have the first bit always 1, such that the number of inversion cases to investigate is reduced. Truth tables having a 0 as the first bit are complemented and the inversion is recorded.

In addition to enumerating small dependency circuits, a special type of node replacement, called *R-resubstitution*, is explored. R-resubstitution exploits the *relevance rule* of majority gates [22]

$$M(x, y, z) = M\left(x_{y/\bar{z}}, y, z\right) \quad (20)$$

where $x_{y/\bar{z}}$ is obtained by replacing all occurrences of y with $\neg z$ in x. Instead of substituting the root node with a dependency circuit in the classical resubstitution, R-resubstitution substitutes a fanin node x of the root $r = M(x, y, z)$ with a divisor d if $(x \oplus d)(y \oplus z) = 0$ and r is the only fanout of x. Unfortunately, finding R-resubstitution cannot be formulated as a resynthesis problem, thus it is not considered in the remainder of this article.

The core problem resubstitution algorithms solve is logic resynthesis. Existing works on resubstitution are based on enumeration, thus there exist small upper bounds on the size of dependency circuits they can find. In contrast, the heuristic resynthesis algorithms proposed in this work are unlimited in this respect.

D. Akers' Majority Synthesis

Akers' majority synthesis algorithm was the earliest work on heuristic synthesis of MIGs [27]. It is a bottom-up approach that builds new gates using the constructed ones. In [27], Akers' Algorithm was presented to synthesize an MIG for any given function from primary inputs, but the algorithm can actually also solve the MIG resynthesis problem. First, the truth tables of the primary inputs are *normalized* by taking their XNOR with the target function, such that the goal of the algorithm becomes building the constant 1 function. The main data structure in Akers' Algorithm, called the *unitized table*, is a collection of the normalized truth tables of primary inputs (and their negations) and of the outputs of MAJ gates created throughout the algorithm. Each column of the unitized table corresponds to a node (a PI or a gate) that can be used to build the next gate, and each row corresponds to a value assignment to the PIs (i.e., a minterm). The algorithm iteratively *reduces* the unitized table, by removing redundant columns and dominated rows, and *expands* the unitized table, by choosing three columns to build a new MAJ gate and adding a new column. The procedure repeats until there is only one column of all 1s left, or until the resource limit exceeds. The choice on using which columns to build new gates is heuristic, so the algorithm does not guarantee to always find a solution.

IV. Heuristic AND-Based Resynthesis

In this section, we introduce the heuristic AND-based resynthesis algorithm which resynthesizes an AIG or an XAG. The algorithm primarily considers AND gates (and cost-free inverters), but it may be extended to consider XOR gates as well, although in a limited way. The algorithm is based on: 1) classification of divisors and 2) recursive decomposition. The former idea has been practically adopted in enumeration-based resubstitution [4], but rarely described in the literature. In Section IV-A, we give the definition of the unateness of divisors and explain why it is useful in reducing the search space of resynthesis. On top of that, in Section IV-C, we propose the recursive decomposition, which is key for our resynthesis algorithm being unbounded by the solution size.

We use figures to illustrate essential concepts in this section. In Figs. 1–3, a rectangle marks the Boolean space under which the target and divisor functions are defined ($\mathbb{B}^k$ in Problem Formulation 1 and 2 or X in Problem Formulation 3). Black dots in the rectangle represent onset minterms of the target and white dots represent offset minterms. In the space where no dots are present, there can be don't-care minterms. For clearer

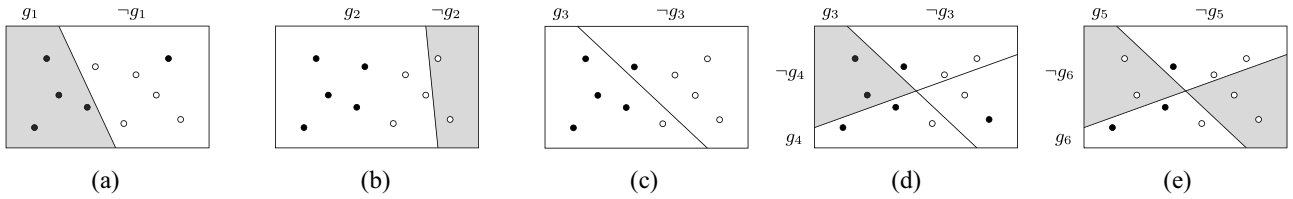

Fig. 1. Illustration of unate literals and binate divisors. (a) Literal g_1 is positive unate. (b) Literal $\neg g_2$ is negative unate. (c) g_3 is a binate divisor. (d) AND-pair $g_3 \wedge \neg g_4$ is positive unate. (e) XOR-pair $g_5 \oplus g_6$ is negative unate.

illustration, don't-care minterms are plotted as gray dashed dots in Fig. 3.

A divisor function g separates the Boolean space into two halves, the region where $g = 1$ and the region where $g = 0$ (or equivalently, $\neg g = 1$). We refer to a divisor with or without negation as a *literal*, i.e., a literal is either a divisor g or a negated divisor $\neg g$, corresponding, respectively, to the two halves of the Boolean space.

A. Classification of Divisors

Any composition of some divisor functions is also a function defined over the same Boolean space, thus also separates the space into two halves. For example, composing two literals l_1 and l_2 with an AND gate results in a separation where the region $l_1 \wedge l_2 = 1$ is the intersection of the regions $l_1 = 1$ and $l_2 = 1$, and the region $l_1 \wedge l_2 = 0$ is the union of the regions $l_1 = 0$ and $l_2 = 0$. The goal of the resynthesis algorithm is to find a composition whose resulting function separates the Boolean space into a half containing only onset minterms of the target and a half containing only offset minterms.

We observe that, if two literals l_1 and l_2 are to be composed using an AND gate and realizing the target, then the regions $l_1 = 0$ and $l_2 = 0$ must not contain any onset minterm of the target. Similarly, if two literals l_3 and l_4 are to be composed using an OR gate (equivalent to an AND gate with input and output negations) and realizing the target, then the regions $l_3 = 1$ and $l_4 = 1$ must not contain any offset minterm of the target because the resulting region $l_3 \vee l_4 = 1$ is the union of the regions $l_3 = 1$ and $l_4 = 1$. We call such property *unateness*.

A literal l is said to be *positive unate* if $l \wedge f_{\text{off}} = 0$. For example, in Fig. 1(a), g_1 is positive unate. Similarly, a literal l is said to be *negative unate* if $l \wedge f_{\text{on}} = 0$. For example, in Fig. 1(b), $\neg g_2$ is negative unate. In contrast to unate literals, binateness is defined for divisors. Given a divisor g, if both g and $\neg g$ are neither positive nor negative unate, then g is said to be a *binate* divisor. For example, in Fig. 1(c), g_3 is a binate divisor. Note that unateness is defined for literals and binateness is defined for divisors. A (nonbinate) divisor g may have one of its literals being unate, but the other literal being neither positive nor negative unate, such as g_1 in Fig. 1(a) and g_2 in Fig. 1(b). Also note that these definitions are different from the unateness of a Boolean function with respect to a variable [28].

Only unate literals can be used to construct the target function using one gate. Thus, by classifying divisors, the number of comparisons required to identify dependency circuits of no more than one gate is reduced. Nevertheless, binate divisors are not totally useless. Two binate divisors may be composed with a gate and become unate. Thus, the definitions of positive and negative unateness are extended for pairs of literals. A pair p of two literals l_1 and l_2 obtained from (optionally negating) two binate divisors is said to be a positive unate *AND-pair* if $(l_1 \wedge l_2) \wedge f_{\text{off}} = 0$. For example, in Fig. 1(d), $(g_3, \neg g_4)$ is a positive unate AND-pair. Similarly, it is negative unate if $(l_1 \wedge l_2) \wedge f_{\text{on}} = 0$. When finding unate pairs, we investigate all pairs of two binate divisors and all of the four possible inverter configurations, corresponding to the four regions of the Boolean space divided by the two divisor functions. There is no need to try an OR-pair because composing two binate divisors with an OR gate (i.e., taking the union) will never lead to a unate function. If XOR gates are allowed, we additionally try to find unate XOR-pairs. For example, in Fig. 1(e), (g_5, g_6) is a negative unate XOR-pair.

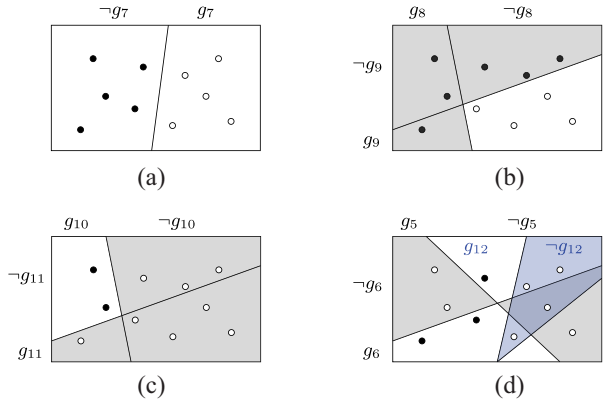

Fig. 2. Illustration of composing simple dependency circuits. (a) $\neg g_7$ is a 0-resyn. (b) $g_8 \vee \neg g_9$ is a 1-resyn. (c) $g_{10} \wedge \neg g_{11}$ is a 1-resyn. (d) $g_{12} \wedge \neg(g_5 \oplus g_6)$ is a 2-resyn.

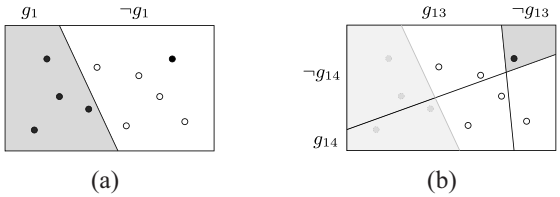

Fig. 3. Illustration of the recursive decomposition. (a) Decompose f_{on} with a positive unate literal g_1. (b) f'_{on} can be more easily realized by $\neg g_{13} \wedge \neg g_{14}$.

B. Simple Dependency Circuits

Simple dependency circuits of no more than three gates are identified similarly to the enumeration-based method. First, if

Algorithm 1: Heuristic AND-Based Resynthesis Algorithm

Input: target onset f_{on}, target offset f_{off}, divisors $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit H

```
1  if f_on = 0 then return Constant 0
2  if f_off = 0 then return Constant 1
3
4  U_p ← positive_unate(G, f_off)
5  U_n ← negative_unate(G, f_on)
6  B ← binate(G, U_p, U_n)
7
8  if u ← find_0resyn(U_p, U_n) then return u
9
10 U_p ← sort(U_p, f_on)
11 U_n ← sort(U_n, f_off)
12
13 if u, v ← find_1resyn(U_p, f_on) then return u ∨ v
14 if u, v ← find_1resyn(U_n, f_off) then return ¬u ∧ ¬v
15
16 P_p ← positive_unate_pair(B, f_off); P_p ← sort(P_p, f_on)
17 P_n ← negative_unate_pair(B, f_on); P_n ← sort(P_n, f_off)
18
19 if p, u ← find_2resyn(P_p, U_p, f_on) then
20     return (p_1 ∘_p p_2) ∨ u
21 if p, u ← find_2resyn(P_n, U_n, f_off) then
22     return ¬(p_1 ∘_p p_2) ∧ ¬u
23 if p, q ← find_3resyn(P_p, f_on) then
24     return (p_1 ∘_p p_2) ∨ (q_1 ∘_q q_2)
25 if p, q ← find_3resyn(P_n, f_off) then
26     return ¬(p_1 ∘_p p_2) ∧ ¬(q_1 ∘_q q_2)
27
28 u ← choose_top(U_p, U_n, P_p, P_n)
29 f'_on ← new_target(u, f_on)
30 f'_off ← new_target(u, f_off)
31 H_r ← resynthesize(f'_on, f'_off, G)
32 return u ∘_u H_r
```

the target onset or offset is empty, then it can be realized with a constant (lines 1 and 2 in Algorithm 1). After classifying divisors and collecting unate literals as described in Section IV-A (lines 4–6), we first check if there exists a literal that realizes the target without extra gates. That is, if a literal l is positive unate and its negation $\neg l$ is negative unate, then l realizes the target (line 8). We call this a 0-*resyn* because it has 0 gates in the dependency circuit. For example, in Fig. 2(a), $\neg g_7$ is positive unate and g_7 is negative unate, thus $\neg g_7$ is a 0-resyn.

To find dependency circuits with one gate, called 1-*resyn*, we try to compose two positive unate literals with an OR gate, or to compose two negative unate literals with an AND gate (lines 13 and 14). For each pair l_1 and l_2 of positive unate literals, we check if their union contains all of the onset minterms. That is, if $\neg(l_1 \vee l_2) \wedge f_{\text{on}} = 0$, or equivalently, $\neg l_1 \wedge \neg l_2 \wedge f_{\text{on}} = 0$. We do not need to check for offset minterms thanks to the definition of positive unate literals. For example, Fig. 2(b) is an OR-type 1-resyn because there is no more onset minterms in the white region. Similarly, two negative unate literals l_3 and l_4 form an AND-type 1-resyn if their union contains all of the offset minterms. That is, $\neg l_3 \wedge \neg l_4$ realizes the target if $\neg l_3 \wedge \neg l_4 \wedge f_{\text{off}} = 0$, such as Fig. 2(c). As the condition to be checked in this step is whether the union of two literals contains all onset (for positive unate) or offset (for negative unate) minterms, we first sort the literals based on how many onset or offset minterms they contain (lines 10 and 11). This way, we may terminate the investigation earlier when we know the remaining pairs of literals all have a total number of onset (or offset) minterms less than the number of onset (or offset) minterms of the target.

If a dependency circuit of size no more than one cannot be found, we proceed to collect unate pairs (lines 16 and 17) and try to find a 2-*resyn* (lines 19–22) or 3-*resyn* (lines 23–26). A 2-resyn is composed of a unate literal and a unate pair. The conditions to be checked are similar to those for 1-resyn. For example, in Fig. 2(d), a negative unate literal $\neg g_{12}$ and a negative unate XOR-pair (g_5, g_6) [taken from Fig. 1(e)] forms an AND-type 2-resyn. Similarly, a 3-resyn is composed of two unate pairs. In Algorithm 1, we use $\circ$ to denote an unspecified gate type depending on the pair noted as the subscript, and we use p_1 and p_2 to denote the two elements of a pair p.

C. Recursive Decomposition

When the target cannot be realized within three gates, the algorithm heuristically chooses a unate literal or a unate pair to decompose the target function (lines 28–32). If a positive unate literal l_1 is chosen, a new target onset $f'_{\text{on}} = f_{\text{on}} \wedge \neg l_1$ with fewer minterms is derived by constructing the dependency circuit with an OR gate on top, having l_1 as one of its fanins. Then, Algorithm 1 is recursively called on the new onset f'_{on} and the same offset $f'_{\text{off}} = f_{\text{off}}$ (line 31) to construct the remaining circuit as the other fanin of the top OR gate. For example, in Fig. 3(a), we decompose f_{on} with a positive unate literal g_1 [taken from Fig. 1(a)], resulting in f'_{on} in Fig. 3(b). The new f'_{on} has only one onset minterm remaining and is more easily realized by $\neg g_{13} \wedge \neg g_{14}$, which were both binate before decomposition. The original target function is thus realized by $g_1 \vee (\neg g_{13} \wedge \neg g_{14})$.[3] In contrast, if a negative unate literal l_2 is chosen, the target onset stays the same, whereas a new offset $f'_{\text{on}} = f_{\text{off}} \wedge \neg l_2$ is derived. The dependency circuit is then constructed with an AND gate with negated fanins on top.

The choice on which literal or pair to use to decompose (line 28) is made by comparing the number of onset (for positive unate literals or pairs) or offset (for negative unate) minterms they contain. The one containing the most minterms is preferred. However, a pair is only chosen if it contains more than twice the number of minterms than the winning literal because choosing a pair leads to one more gate in the dependency circuit.

D. Summary

Algorithm 1 summarizes the AND-based resynthesis algorithm. In Algorithm 1, lines 1–26 are similar to enumeration-based resubstitution, which resynthesizes dependency circuits of at most 3 gates. Lines 28–32 are the key for the algorithm

[3]The example is made simple for easier understanding. This solution can actually be found as a 2-resyn without the recursive decomposition. To give a real example where recursive decomposition is needed, for example, g_{13} and g_{14} could be pairs instead of divisors, which only become unate with respect to the new onset f'_{on}.

to resynthesize larger dependency circuits, where line 31 calls the resynthesis algorithm recursively.

It is neglected in the pseudocode, but in practice an additional parameter *size limit* is passed to the algorithm. Before each step, the size limit is checked and the algorithm terminates without a solution if the limit is reached. For example, before *find_3resyn*, if *size limit* is 2, the algorithm returns *no solution*. In line 31, the *size limit* being passed to the recursive call is the current size limit minus 1 (when decomposing with a literal) or 2 (when decomposing with a pair). When the algorithm returns *no solution*, it is possible that a solution larger than *size limit* exists and can be found if *size limit* were set larger, or that the given problem is infeasible. It is also possible that a solution exists, but cannot be found by the algorithm because it is heuristic, irrelevant to *size limit*. The same early-termination mechanism also applies to the following MAJ-based and MUX-based resynthesis algorithms.

V. Heuristic MAJ-Based Resynthesis

We introduce the heuristic MAJ-based resynthesis algorithm in this section, based on the following key ideas.

1) *Normalization:* Divisor functions are normalized to simplify the algorithm and reduce the number of bitwise operations needed. This step is done only once in the beginning (Section V-A).
2) *Covering the Care Function:* We introduce the notion of *care functions* at any position in the dependency circuit under construction. The goal of the algorithm is to *cover* more uncovered bits in the care function by modifying the current dependency circuit until all bits are covered (Section V-B).
3) *Heuristic Choice of Divisors:* The algorithm repeatedly chooses three divisors to form a new majority gate. Divisors are chosen according to their evaluation on a heuristic weight function with respect to the current care function (Section V-C).
4) *Expansion to a Tree-Like Circuit:* The algorithm constructs the dependency circuit by repeatedly expanding on a leaf of the circuit. It chooses a fanin of a gate which is connected to a divisor, takes out the divisor, and replaces it with a newly constructed gate. The resulting circuits thus have tree-like structures (Section V-D).

A. Normalization

Given the target f and the set of divisors $G = \{g_1, \ldots, g_n\}$, the divisors are *normalized* by computing their XNOR with the target. By doing so, the logic of the algorithm is simplified—comparing the output function of the dependency circuit against the target simplifies to testing if the output function is a tautology. Moreover, due to the self-duality property of the majority function [23], inverters can always be pushed to the primary inputs. Hence, we limit our search to dependency circuits without internal inverters and consider inverters only at the inputs by supplementing the divisor set with negated literals. The set N of normalized literals to be chosen from as inputs to the dependency circuit is computed by

$$N = \{l_{2i-1} = g_i \leftrightarrow f,\ \ l_{2i} = \neg g_i \leftrightarrow f \mid 1 \le i \le n\}. \quad (21)$$

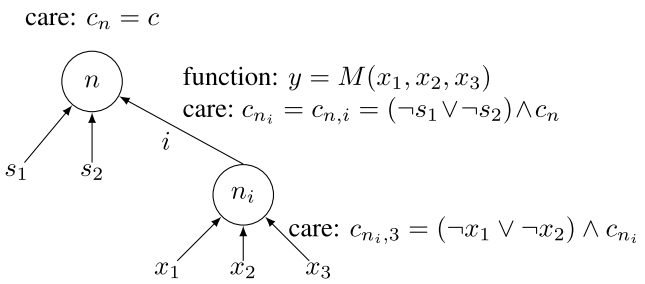

Fig. 4. Illustration of the care functions.

B. Care Function

Consider an MAJ gate with function $y = M(x_1, x_2, x_3)$ and a certain bit position p in its truth table. In order to have $T[y]_p = 1$, we must have

$$T[x_i]_p = T\big[\overline{x_j}\big]_p = 1, \ \text{where } i, j \in \{1, 2, 3\} \text{ and } i \neq j.$$

If the functions x_1 and x_2 have been decided but x_3 is still flexible, then, we require $T[x_3]_p = 1$ only if $T[x_1]_p = 0$ or $T[x_2]_p = 0$. In such case, we say that p is a *care bit* for the third fanin of the gate under construction.

Generalizing and extending to all bit positions, we define the *care function* c_i of a fanin i to a node n as

$$c_{n,i} = (\neg s_1 \vee \neg s_2) \wedge c_n \quad (22)$$

where s_1 and s_2 are the other two fanin functions of n (i.e., siblings of i) and c_n is the care function of n. If n is the topmost node of the dependency circuit, as in Fig. 4, then its care function c_n is the care function c of the target, given as input to the resynthesis problem. Otherwise, as our dependency circuits are tree-like, the node n must have exactly one fanout (parent) node, and its care function is derived using (22) according to its parent's care function and its siblings' functions. For example, the care function c_{n_i} of node n_i in Fig. 4 is the care function of the fanin i to node n.

A care bit in a care function is said to be *covered* if the function presented at the node (or at the fanin edge) indeed provides 1 at this bit. For example, for a care bit in $c_{n_i,3}$ to be covered, the function x_3 needs to be 1 at this bit. If the care function of a node (e.g., c_{n_i} in Fig. 4) is of interest, then, we need at least two fanin functions of the node (e.g., x_1 and x_3) to *cover* the bit by having 1's.

C. Choosing Divisors

Given the care function c_n of a node n, a heuristic selection is used to choose three literals l_1, l_2, and l_3 from N to construct an MAJ gate, aiming at maximizing $\textsc{Ones}(M(l_1, l_2, l_3) \wedge c_n)$

$$\begin{aligned} l_1 &= \operatorname*{argmax}_{l \in N}(\textsc{Ones}(l \wedge c_n)) \\ l_2 &= \operatorname*{argmax}_{l \in N_2}(\textsc{Ones}(l_1 \wedge l \wedge c_n) + 2 \cdot \textsc{Ones}(\neg l_1 \wedge l \wedge c_n)) \\ l_3 &= \operatorname*{argmax}_{l \in N_3}(\textsc{Ones}((l_1 \oplus l_2) \wedge l \wedge c_n) \\ &\qquad + 2 \cdot \textsc{Ones}((\neg l_1 \wedge \neg l_2) \wedge l \wedge c_n) \end{aligned}$$

$$\text{where } N_2 = N \backslash \{l_1, \neg l_1\}, N_3 = N_2 \backslash \{l_2, \neg l_2\}. \quad (23)$$

The first literal is chosen to cover most care bits. When choosing the second literal, the care bits covered by the first

literal still need to be covered again, thus we acknowledge more $\text{ONES}(l_1 \wedge l \wedge c_n)$. But more importantly, we are more eager to cover the care bits that are not covered by the first literal, thus the weight for $\text{ONES}(\neg l_1 \wedge l \wedge c_n)$ is doubled. For the last literal, the care bits that are already covered twice can be ignored; the care bits covered only once $((l_1 \oplus l_2) \wedge l \wedge c_n)$ seek to be covered again; the care bits that are never covered before $((\neg l_1 \wedge \neg l_2) \wedge l \wedge c_n)$ appear to be more difficult to cover than the other bits and they are thus doubly weighed. In the last case, it may seem counter-intuitive to cover these bits with the last literal because covering them only once is not enough. However, the first two literals may be replaced by new nodes later on in the algorithm, so it is still useful to cover them at least once in this stage.

This evaluation step will be repeatedly incurred throughout the algorithm. The computational complexity is linear to the number of divisors, which can be large. We observe that the resulting choice depends solely on the care function c_n. To speed up the computation, a computed table can be used to cache the results. This is implemented as a hash table mapping from a care function to three divisors.

D. Expansion

When all care bits of the three fanins of the topmost node are covered, the constant 1 function is successfully derived at its output and the algorithm terminates. After constructing the first node with three literals, we choose one of the fanins with uncovered care bits, if any, and try to cover more care bits by replacing the literal with a new gate. This process is called an *expansion*.

To *expand* a fanin, the original literal is temporarily taken away. Then, three literals are chosen as the fanins of the new gate using (23). After an expansion, the function at the expanded fanin is different, thus the functions of its transitive fanouts, as well as the care functions of its siblings, are updated accordingly. Until the constant 1 is derived at the output of the topmost node by covering all the care bits, the algorithm proceeds by choosing another position to expand. An *expansion position* is a fanin of any node which is connected to a literal and whose care function is not fully covered. Heuristically, we choose the position with the least uncovered care bits to be expanded first because it is closest to be fully covered.

It is possible that the majority output of the three chosen literals does not cover more care bits than the original literal. Hence, the new gate is only constructed and used to replace the original literal if the number of covered care bits increases. When an expansion position is tried but the coverage of care bits does not increase, the new gate is discarded and the position is marked as visited to avoid trying it again. However, if its care function is updated because of an update in the function of one of its siblings, the visited flag is reset and the expansion position may be tried again. To avoid constructing gates using the same literals repeatedly as a chain, when the care function of a node is the same as one of its fanins, the expansion position at this fanin is directly marked as visited without trying to expand it.

Algorithm 2: Heuristic MAJ-Based Resynthesis Algorithm

Input: target function f, care function c, divisor functions $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit H

```
1  N ← normalize(G, f)
2  n_0 ← choose_literals(N, c)
3  H ← {n_0}
4  while n_0.output ≠ 1 do
5      (n_p, i) ← choose_expansion_position(H)
6      n ← choose_literals(N, n_p.fanin(i).care)
7      if accept_expansion(n_p, i, n) then
8          n_p.fanin(i) ← n
9          update(H)
10     else
11         mark_visited(n_p, i)
12 return H
```

E. Summary and Example

Algorithm 2 summarizes the heuristic MAJ-based resynthesis algorithm. First, the set of divisors is normalized and supplemented using (21) (line 1). Then, the top node n_0 is constructed by choosing three literals using (23) and added into the dependency circuit as the first node (lines 2 and 3). If the output function of n_0 is not constant 1 (line 4), we choose an expansion position (the ith fanin of a parent node n_p) which is currently connected to a literal (line 5). The care function of the position is computed by (22) and used to choose three literals to construct a new gate (line 6). If replacing the original literal with the new gate increases the number of covered care bits, the expansion is accepted and the dependency circuit is updated (lines 7–9); otherwise, the position is marked as visited (lines 10 and 11). The expansion procedure is repeated until the constant 1 function is obtained at the output of the topmost node.

An example execution of the algorithm is illustrated in Fig. 5, where the target function is

$$f(\vec{x}) = x_1 \oplus x_2 \oplus x_3 \tag{24}$$

the care function $c = 1$, and the set G of divisors consists of

$$G = \{g_1(\vec{x}) = x_1, g_2(\vec{x}) = x_2, g_3(\vec{x}) = x_3, g_4(\vec{x}) = 0\}. \tag{25}$$

The normalized set N of literals, computed according to (21), is listed in their truth table representations in the box in Fig. 5(a). The yellow-shaded parts Fig. 5 are the truth tables being updated after expansions. First, in Fig. 5(a), given the care function $c = 1$, three literals l_7, l_1, and l_3 are chosen according to (23) to form the topmost node n_0, computing the function at its output $n_0 = M(l_7, l_1, l_3)$. Care functions of each fanin $c_{0,i}$ are computed according to (22). Then, in Fig. 5(b), the first fanin of n_0 is chosen to be expanded with a new node n_1. According to its care function $c_{0,1}$, three literals l_2, l_4, and l_6 are chosen. The function at the expanded fanin is updated with $n_1 = M(l_2, l_4, l_6)$. Following which, the care functions at its siblings $c_{0,2}$ and $c_{0,3}$, as well as the output function n_0 are also updated. After the expansion, all care bits of the first fanin of n_0 have been covered by the function of n_1, but there are still

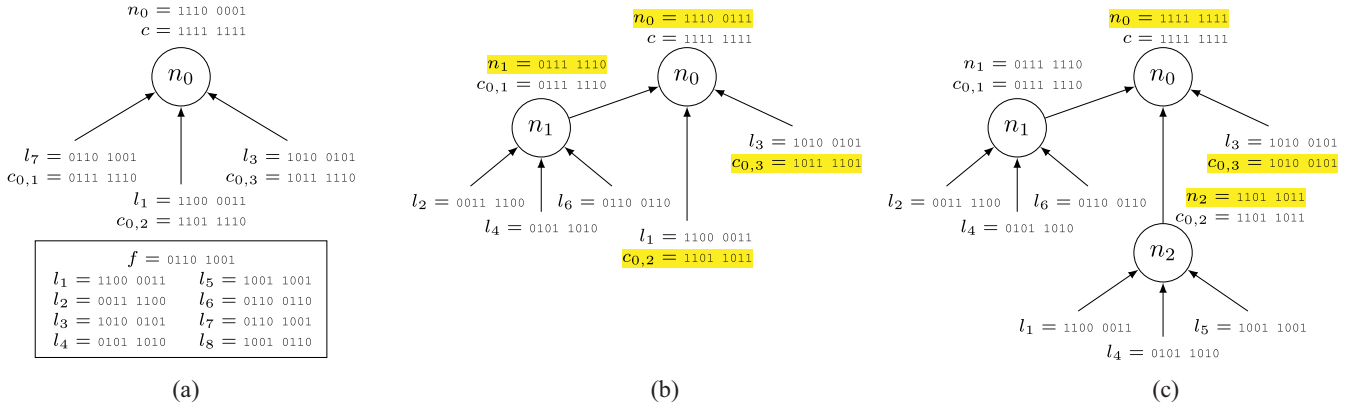

Fig. 5. Example of MAJ-based resynthesis. (a) Topmost node n_0. (b) Expand at $(n_0, 1)$ with $n_1 = M(l_2, l_4, l_6)$. (c) Expand at $(n_0, 2)$ with $n_2 = M(l_1, l_4, l_5)$.

two care bits in each of the updated $c_{0,2}$ and $c_{0,3}$ not yet covered. So, in Fig. 5(c), the second fanin of n_0 is expanded with another new node n_2. Similarly, according to its care function $c_{0,2}$, three literals l_1, l_4, and l_5 are chosen, and the node functions n_2 and n_0, as well as the sibling's care function $c_{0,3}$, are updated. Now, all care bits in $c_{0,2}$ and also $c_{0,3}$ are covered, and the output function of n_0 is constant 1. The resynthesis has thus been completed. The final solution is $h(g_1, g_2, g_3, g_4) = M(M(\neg g_1, \neg g_2, \neg g_3), M(g_1, \neg g_2, g_3), g_2)$.

VI. Heuristic MUX-Based Resynthesis

Although rarely researched on, MuxIGs may be a practical data structure for some technologies where MUX gates are of similar cost as AND and XOR gates, such as memristors [29], quantum-dot cellular automata (QCA) [30], and pass transistor logic [31]. Although the MUX gate itself is functionally complete without inverters, we still use complemented edges to represent cost-free inverters in the network to be more memory efficient. This can be disabled [i.e., $\neg x$ has to be implemented as MUX$(x, 0, 1)$] and the MUX-based resynthesis algorithm can also be adjusted accordingly, if desired. A MUX gate can implement the 2-input AND, OR, and XOR functions, thus MuxIGs are more compact than XAGs. Though conceptually similar, MuxIGs are different from BDDs [16]. In BDDs, S-inputs can only be primary variables, whereas in MuxIGs, S-inputs can be connected to the output of any other MUX gates in the network. Thus, MuxIGs are more general than BDDs. In this section, we propose a MUX-based resynthesis algorithm that can be used to optimize MuxIGs.

Due to the natural characteristics of the MUX gate, our MUX-based resynthesis algorithm is designed with a combination of ideas from AND- and MAJ-based resynthesis. First, we observe that, similar to resynthesizing with MAJ gates, we seek to select or construct functions resembling the target to be placed at the T- and E-inputs of a MUX gate, subject to a care function depending on the function at its S-input. Thus, we also normalize divisor functions and adopt the bit-counting-based ranking and selection of divisors as in MAJ-based resynthesis. Second, when there are some care bits not covered, unlike MAJ-based resynthesis, the expansions on the T- and E-inputs are independent of each other. For a MUX gate with care function c, once the S-input s is selected, the care function at the T-input is $c_t = c \wedge s$ and the care function at the E-input is $c_e = c \wedge \neg s$. Thus, we adopt the recursive decomposition similar to that in AND-based resynthesis to expand on T- or E-inputs until all care bits are covered. To avoid renormalizing divisors and to simplify the computation, we do not expand on the S-input once it is selected.

Algorithm 3: Heuristic MUX-Based Resynthesis Algorithm

Input: target function f, care function c, divisor functions $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit H

```
1  N ← normalize(G, f)
2  return resynthesize(c)
3
4  Function resynthesize(care c):
5      t ← argmax_{l∈N} ONES(l ∧ c)
6      if ONES(¬t ∧ c) = 0 then
7          return t
8      S ← argmin_{l∈{g,¬g : g∈G}} ONES(¬t ∧ l ∧ c)
9      s ← argmin_{l∈S} ONES(¬l ∧ c)
10     if ONES(¬s ∧ c) = 0 then
11         e ← 0
12     else
13         e ← argmax_{l∈N} ONES(l ∧ ¬s ∧ c)
14         if ONES(¬e ∧ ¬s ∧ c) > 0 then
15             e ← resynthesize(¬s ∧ c)
16     if ONES(¬t ∧ s ∧ c) > 0 then
17         t ← resynthesize(s ∧ c)
18     return MUX(s, t, e)
```

Algorithm 3 illustrates the MUX-based resynthesis algorithm. First, the set N of normalized divisors is derived using (21) (line 1). The unchanged set N is then available and used throughout the algorithm along with the original set of divisors G. The recursive algorithm starts with the given top-level care function c (line 2). In line 5, a literal t covering the most care bits is chosen from N as the T-input. If all care bits are covered by t, then it is a 0-resyn and is returned (lines 5 and 6). Otherwise, we continue to choose a literal s from G as the S-input using two criteria: literals S whose (cared) 1-bits overlap the least with the 0-bits of t are prioritized (line 8). If there are more than one literal in S, then the literal with the

least 0 in the cares bits is chosen (line 9). The first criterion aims at reducing uncovered care bits at the T-input, whereas the second criterion aims at reducing the care bits to be covered at the E-input. If the selected s has no cared 0-bit, then the function at the E-input does not matter and we choose constant 0 as the E-input, assuming it has the lowest cost (lines 10–11). Otherwise, similar to choosing t, a literal e covering the most care bits is chosen as the E-input (line 13). Although the philosophy behind the choice of t and the choice of e is the same, there is a difference in their evaluations. When choosing t, the S-input is not selected yet, thus only the care function c for the gate is considered. However, when choosing e, the S-input s is already decided, thus the more precise care function at the E-input $c_e = c \wedge \neg s$ is considered. Finally, we check if the care bits at the T- and E-inputs are all covered by t and e, respectively, and recursively expand on the inputs using their care functions if not so (lines 16 and 17 and 14 and 15, respectively).

VII. Experimental Results

The three resynthesis algorithms are implemented in C++ as part of the logic synthesis library *mockturtle*[4] [32]. In this section, we test the performance and efficiency of the proposed resynthesis algorithms on sets of real resynthesis problems extracted from the EPFL benchmarks [33] by resubstitution (Section VII-A). We also demonstrate in Section VII-B the effectiveness of using resynthesis as the core of a high-effort optimization to further optimize highly optimized benchmarks. The experiments were conducted on a laptop with Apple M1 Pro chip and 32-GB RAM.

A. Extracted Resynthesis Problems

As the core of peephole optimization, it is more meaningful to compare different resynthesis approaches using real resynthesis problems in their general form, with arbitrary divisor functions coming into play. In this section, we test our heuristic resynthesis algorithm on sets of resynthesis problems extracted from the EPFL benchmark suite. The benchmarks are preprocessed by running the script `compress2rs` in ABC [34] once to rule out most optimizations that are easier to identify. To extract resynthesis problems, for each node (root) in the benchmarks, a reconvergence-driven cut [4] of size $k = 4$ or 6 is computed and used as the basis to obtain local functions of nodes supported by the cut. The function of the root node is the target of the resynthesis problem and the functions of all nodes supported by the cut, including the cut leaves, are divisors. The care set is derived by computing (local) satisfiability don't cares from a larger cut of size 12. A size limit $\max m$ is given along with the resynthesis problem, determined by the size of the root's *maximum fanout-free cone* (MFFC) [35] minus 1.

Three sets of AIG resynthesis problems are considered in Table I.

1) *First Big Column:* A subset of problems extracted using cut size $k = 4$ (thus truth table length $l = 2^k = 16$) where the size limit is at least 1.
2) *Second Big Column:* A subset of problems extracted using cut size $k = 6$ (thus truth table length $l = 2^k = 64$) where the size limit is at most 3.
3) *Third Big Column:* A subset of problems extracted using cut size $k = 6$ where the size limit is at least 4.

[4]Available: https://github.com/lsils/mockturtle.

TABLE I
Comparison of AIG Resynthesis Algorithms

	1) $k = 4$, $\max m \geq 1$		2) $k = 6$, all problems		3) $k = 6$, $\max m \geq 4$	
#Probs	128312		337155		22691	
Avg. n	6.55		14.16		7.18	
Avg. $\max m$	1.53		0.70		4.18	
	SAT	Ours	Enum.	Ours	SAT	Ours
#Sols	920	990	1248	1589	522	465
Avg. m	1.72	1.71	1.97	2.61	4.17	4.38
Avg. overhead	–	0.00 (0%)	–	0.05 (1%)	–	0.16 (3%)
Tot. time (s)	43.12	0.11	0.28	0.34	638.21	0.10

The total number of resynthesis problems ("#Probs"), the average number of divisors per problem ("Avg. n"), and the average size limit ("Avg. $\max m$") are listed for each set in the upper half of Table I. We compare our AND-based heuristic resynthesis ("Ours") against SAT-based exact synthesis [26] ("SAT," Section III-B, conflict limit = 10 000) and enumeration-based method [4] ("Enum.," Section III-C, up to 3 gates). The number of solutions found within the size limit ("#Sols"), the average number of gates in the dependency circuits found ("Avg. m"), the average overhead comparing to the optima ("Avg. overhead"), and the total runtime in seconds ("Tot. time") are listed for each method.

We observe from this experiment that resynthesis problems requiring larger dependency circuits do exist in real benchmarks. Both SAT and enumeration are exact algorithms, meaning that the solutions they give, if any, are always optimal. However, the optimality of SAT-based exact synthesis comes with the cost of a much higher runtime, and enumeration, although being fast, can only solve problems with small solutions. In 2), the 341 more problems solved by our heuristic than enumeration are cases where a solution cannot be found within three gates and the recursive decomposition described in Section IV-C is necessary. The quality degradation of our heuristic is zero for smaller dependency circuits ($m \leq 3$) and is still very small (3%) for medium-sized dependency circuits for which SAT-based synthesis needs a long time to find the optimal solution.

B. Resynthesis as the Core of High-Effort Optimization

To demonstrate the practical application of the proposed heuristic resynthesis algorithms in high-effort optimization, we use them as the core component in the simulation-guided resubstitution framework [12] and perform experiments on benchmarks that are already optimized by state-of-the-art size optimization flows. The resubstitution framework computes, for each target node as the root, a reconvergence-driven cut of at most 8 nodes to collect up to 150 divisors supported by the

TABLE II
AND-BASED HEURISTIC RESYNTHESIS AS THE CORE OF SIMULATION-GUIDED RESUBSTITUTION APPLIED ON HIGHLY OPTIMIZED AIG BENCHMARKS

AIG: O = Original, A = `compress2rs` ×1, B = `compress2rs` ×∞								
Benchmark		O → A		A → Ours		A → B	B → Ours	
Name	Size (#gates)	Red. (%)	Time (s)	Red. (%)	Time (s)	Red. (%)	Red. (%)	Time (s)
adder	1020	12.55	0.08	0.00	0.00	0.00	0.00	0.00
bar	3336	5.85	0.27	2.58	0.04	0.00	2.58	0.04
div	57247	63.80	3.64	0.89	0.23	1.05	0.00	0.63
hyp	214335	4.57	30.03	0.15	14.93	0.16	0.05	14.44
log2	32060	8.95	5.08	1.73	5.41	0.56	1.54	6.10
max	2865	1.15	0.18	0.00	0.01	0.28	0.00	0.02
multiplier	27062	10.07	3.53	0.10	0.28	0.09	0.01	0.27
sin	5416	7.33	0.97	1.35	0.44	1.00	1.19	0.58
sqrt	24618	25.85	2.87	0.30	4.39	0.01	0.26	4.39
square	18484	14.03	2.61	0.66	0.13	0.57	0.09	0.06
arbiter	11839	0.00	1.42	0.00	0.15	0.00	0.00	0.28
cavlc	693	8.37	0.19	4.25	0.09	2.20	3.06	0.16
ctrl	174	48.28	0.04	0.00	0.00	0.00	0.00	0.00
dec	304	0.00	0.06	0.00	0.00	0.00	0.00	0.00
i2c	1342	20.34	0.12	2.90	0.02	5.05	2.17	0.02
int2float	260	19.62	0.05	0.96	0.03	0.96	0.48	0.06
mem_ctrl	46836	6.22	5.21	15.95	1.76	12.13	14.09	2.03
priority	978	52.35	0.07	0.64	0.00	8.15	0.23	0.01
router	257	28.79	0.04	20.77	0.00	20.77	9.66	0.00
voter	13758	42.24	1.58	0.18	0.02	0.13	0.08	0.05
Average		19.02	2.90	2.67	1.40	2.66	1.77	1.46
Total gain		71402		8460		6360	6257	

cut. Functions of the target and divisor nodes are estimated by global simulation using about 1000 simulation patterns.

1) AIG: For AIG size optimization, the script `compress2rs` in ABC [34] is considered as the state-of-the-art flow, which comprises 18 commands, including balancing, resubstitution, rewriting, and refactoring with different hyperparameters. In Table II, after listing the benchmark names and their original size, the size reduction in terms of percentage number of gates ("Red.") and runtime ("Time") of four optimization settings are presented. Column "O → A" applies `compress2rs` once on the original benchmarks; we call the resulting set of optimized benchmarks A. Column "A → Ours" applies simulation-guided resubstitution using our heuristic AND-based resynthesis on the benchmark set A. Column "A → B" applies more times of `compress2rs` on A until no more size reduction is observed for at least five consecutive times; we call this set of benchmarks B. Column "B → Ours" applies our resubstitution on the benchmark set B. In the last row, "Total gain" lists the total number of reduced gates, summed over all benchmarks.

Comparing "A → Ours" and "A → B," we can observe that, on top of the benchmark set A that is already optimized, our high-effort optimization achieves similar "leftover" size reduction as the best `compress2rs` can do. Moreover, column "B → Ours" shows that our approach can still squeeze 1.78% more size reduction out of the highly optimized benchmark set B. In both "A → Ours" and "B → Ours," the runtime of our high-effort optimization is comparable with `compress2rs`.

Experiments on XAG, MIG, and MuxIG optimization all use the optimized benchmark set A as the starting point (column "AIG" in Table III). Besides size reduction percentage ("Red.") and total runtime ("Time"; for Columns XAG and MIG, time for `compress2rs` is excluded), the runtime spent by our heuristic algorithms in solving the resynthesis problems is also listed ("T_{resyn}").

2) XAG: For XAG optimization, we first apply the LUT mapping command `&if` in ABC with K (number of inputs per LUT) set to 2, followed by the interpolation-based LUT resubstitution command `&mfs` [8] to obtain XAG benchmarks (column "XAG" in Table III; note that a 2-LUT network is essentially an XAG). Then, in column "XAG → Ours" we apply simulation-guided resubstitution using our AND-based resynthesis with XOR enabled, and 2.86% size reduction is obtained from the set of optimized XAGs within similar runtime as optimizing and transforming into XAGs.

3) MIG: As the state-of-the-art MIG optimization flow, we apply three times graph (re)mapping [36] from the optimized AIGs, followed by enumeration-based MIG resubstitution [10] repeated until no more size reduction is observed (column "MIG" in Table III). Then, similarly, simulation-guided resubstitution using our MAJ-based resynthesis is applied, which obtains 2.45% size reduction on top of highly optimized benchmarks within a faster runtime (column "MIG → Ours" in Table III).

4) MuxIG: Finally, as there is not yet much research on MuxIG, we transform the optimized AIGs directly into MuxIGs by replacing AND gates with MUX gates with a constant input. Then, in column "MuxIG, ours," simulation-guided resubstitution using our MUX-based resynthesis successfully reduces the sizes of these MuxIGs by 20.24% by identifying MUX functions in the networks. It is worth noting that although the runtime for the largest benchmark *hyp* seems to be long, the time spent in the resynthesis algorithm takes only 1% and most of the time is spent in proving the validity of the identified optimization choices.

VIII. CONCLUSION

In this article, three heuristic resynthesis algorithms are proposed, targeting networks based on AND, MAJ, and MUX gates. The common characteristic of the proposed algorithms is that they are efficient heuristics without superlinear scalability concerns. Table IV compares the proposed heuristics with other existing methods. All methods compared solve the resynthesis problem with incompletely specified functions (Problem Formulation 3), except for looking up in an optimal database, which only solves a subset of resynthesis problems where divisors are projection functions and all functions are completely specified. All algorithms are sound, but only database look up, SAT-based exact synthesis, and enumeration are complete and guarantee optimality. As a compromise, these exact methods have a rather high complexity (except for database) and are practically limited by the number of divisors (n), the size of dependency circuit (m), and/or the truth table length (l). In contrast, although the proposed heuristics do not guarantee optimality, their complexities are linear in all variables (or only quadratic in n for AND-based resynthesis) and are thus practically unlimited.

TABLE III
HEURISTIC RESYNTHESIS AS THE CORE OF SIMULATION-GUIDED RESUBSTITUTION APPLIED ON HIGHLY OPTIMIZED BENCHMARKS

AIG = `compress2rs`, XAG = `compress2rs; &if -K 2; &mfs`, MIG = `compress2rs` + map ×3 + resub ×∞														
	AIG	XAG		XAG → Ours			MIG		MIG → Ours			MuxIG, ours		
Benchmark	Size (#gates)	Size (#gates)	Time (s)	Red. (%)	Time (s)	T_{resyn} (s)	Size (#gates)	Time (s)	Red. (%)	Time (s)	T_{resyn} (s)	Red. (%)	Time (s)	T_{resyn} (s)
adder	892	637	0.04	0.00	0.00	0.00	384	0.11	0.00	0.00	0.00	28.48	0.03	0.01
bar	3141	3141	1.16	2.10	0.04	0.03	2594	0.29	0.23	0.03	0.03	43.36	0.07	0.02
div	20725	16791	0.13	0.40	0.63	0.06	12565	0.93	0.26	0.32	0.11	39.24	2.64	0.10
hyp	204533	160201	72.60	5.03	47.55	0.46	127877	13.01	2.89	9.10	0.86	21.56	104.69	1.05
log2	29192	23966	19.58	1.55	2.15	0.22	23643	3.00	2.26	6.41	0.34	14.92	21.23	0.24
max	2832	2832	0.12	0.00	0.02	0.01	2210	0.32	0.00	0.03	0.03	28.32	0.08	0.02
multiplier	24337	18571	10.59	0.12	0.23	0.13	18700	1.76	1.39	0.34	0.20	19.13	4.51	0.20
sin	5019	4263	11.37	2.18	0.54	0.04	4018	0.81	1.27	0.19	0.07	15.06	0.77	0.05
sqrt	18255	14381	0.13	12.79	3.41	0.05	12513	1.09	0.72	3.25	0.16	20.36	4.35	0.11
square	15891	12450	9.80	0.10	0.07	0.04	9573	1.03	0.78	0.08	0.05	30.87	1.06	0.08
arbiter	11839	11839	29.94	0.00	0.34	0.13	6866	1.38	2.14	0.17	0.14	1.08	0.42	0.33
cavlc	635	634	0.12	5.21	0.23	0.22	541	0.83	1.48	0.02	0.02	14.02	0.02	0.01
ctrl	90	90	0.01	4.44	0.00	0.00	80	0.21	1.25	0.01	0.01	15.56	0.00	0.00
dec	304	304	0.01	0.00	0.00	0.00	304	0.09	0.00	0.01	0.01	0.00	0.01	0.01
i2c	1069	1062	0.08	3.48	0.03	0.02	951	0.12	2.00	0.02	0.02	19.36	0.02	0.01
int2float	209	208	0.02	2.88	0.05	0.04	190	0.09	4.74	0.01	0.01	12.44	0.00	0.00
mem_ctrl	43924	38241	61.50	10.11	2.28	1.04	38179	3.86	8.91	2.24	1.24	23.23	2.78	0.97
priority	466	443	0.07	1.13	0.02	0.01	449	0.10	4.01	0.01	0.01	13.30	0.01	0.00
router	183	143	0.01	5.59	0.01	0.00	170	0.07	11.18	0.00	0.00	21.86	0.00	0.00
voter	7946	5717	4.23	0.12	0.53	0.02	4729	0.53	3.55	0.05	0.03	22.73	0.29	0.04
Average			11.08	2.86	2.91	0.13		4.38	2.45	1.11	0.17	20.24	7.15	0.16

TABLE IV
COMPARISONS OF EXISTING AND PROPOSED RESYNTHESIS ALGORITHMS

	Database [5]	SAT-based (SSV encoding) [7], [26]	Enumeration [4], [9], [10]	Akers' [27]	Proposed heuristics
Support of divisors	no	yes	yes	yes	yes
Support of incomplete functions	no	yes	yes	yes	yes
Soundness	yes	yes	yes	yes	yes
Completeness	yes	yes	yes	no	no
Optimality	yes	yes (if solved iteratively)	yes	no	no
Complexity	$\mathcal{O}(1)$	#vars: $\mathcal{O}(m((n+m)^\kappa+l))$ #clauses: $\mathcal{O}(m(n+m)^\kappa)$	$\mathcal{O}(n^{(\kappa-1)m+1}l)$	$\mathcal{O}(n^2ml^2)$	AND-based: $\mathcal{O}(n^2ml)$ MAJ- and MUX-based: $\mathcal{O}(nml)$
Practical limits	$n=k\leq 4$	$n+m\leq 10, k\leq 6$	$m\leq 3$	unknown	no limit

n: number of divisors, m: number of gates in dependency circuit, k: number of variables of target and divisor functions, $l=2^k$: length of truth tables, κ: number of fanins per gate ($\kappa=2$ for AIG and XAG; $\kappa=3$ for MIG and MuxIG)

Experimental results show that the proposed heuristic resynthesis serve as an important component in high-effort peephole optimization, achieving, on average, about 2%–3% more size reduction on benchmarks that are already highly optimized, within manageable runtime. The key to finding these hidden optimization opportunities is the heuristics' capability to solve resynthesis problems with more divisors (scalability in *n*), having larger solutions (scalability in *m*), and where functions are given as longer simulation signatures (scalability in *l*).

ACKNOWLEDGMENT

The authors would like to thank Dr. Heinz Riener and Dr. Alan Mishchenko for their valuable discussions.

REFERENCES

[1] J. A. Darringer and W. H. Joyner Jr., "A new look at logic synthesis," in *Proc. 17th Design Autom. Conf.*, 1980, pp. 543–549.

[2] J. A. Darringer, W. H. Joyner Jr., C. L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM J. Res. Dev.*, vol. 25, no. 4, pp. 272–280, 1981.

[3] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 6, no. 6, pp. 1062–1081, Nov. 1987.

[4] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, 2006, pp. 1–8.

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

[6] 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 Europe Conf. Exhibit.*, 2019, pp. 1649–1654.

[7] H. Riener, A. Mishchenko, and M. Soeken, "Exact DAG-aware rewriting," in *Proc. Design, Autom. Test Europe Conf. Exhibit.*, 2020, pp. 732–737.
[8] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 34:1–34:23, 2011.
[9] L. G. Amarù et al., "Improvements to Boolean resynthesis," in *Proc. Design, Autom. Test Europe Conf. Exhibit.*, 2018, pp. 755–760.
[10] H. Riener, E. Testa, L. G. Amarù, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. 14th IEEE/ACM Int. Symp. Nanoscale Archit.*, 2018, pp. 157–162.
[11] W. Haaswijk, L. G. Amarù, P. Vuillod, J. Luo, M. Soeken, and G. De Micheli, "Integrated ESOP refactoring for industrial designs," in *Proc. 25th IEEE Int. Conf. Electron., Circuits Syst.*, 2018, pp. 369–372.
[12] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 8, pp. 2573–2586, Aug. 2022.
[13] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, "Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis," in *Proc. 27th Asia–South Pacific Design Autom. Conf.*, 2022, pp. 395–402.
[14] S.-Y. Lee, H. Riener, and G. De Micheli, "Logic resynthesis of majority-based circuits by top-down decomposition," in *Proc. 24th Int. Symp. Design Diagnost. Electron. Circuits Syst.*, 2021, pp. 105–110.
[15] D. E. Knuth, *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Boston, MA, USA: Addison-Wesley, 2011.
[16] S. B. Akers Jr., "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 509–516, Jun. 1978.
[17] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture—A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann, 1999.
[18] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1997, pp. 78–82.
[19] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proc. DAC*, 2001, pp. 103–108.
[20] Z. Chu, M. Soeken, Y. Xia, and G. De Micheli, "Functional decomposition using majority," in *Proc. ASP-DAC*, 2018, pp. 676–681.
[21] Y.-T. Lai, K.-R. R. Pan, and M. Pedram, "OBDD-based function decomposition: Algorithms and implementation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 8, pp. 977–990, Aug. 1996.
[22] L. G. Amarù, P. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.
[23] S. Muroga, I. Toda, and S. Takasu, "Theory of majority decision elements," *J. Franklin Inst.*, vol. 271, no. 5, pp. 376–418, 1961.
[24] K. A. Bartlett et al., "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.
[25] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *Proc. Int. Conf. Comput.-Aided Design*, 2007, pp. 227–233.
[26] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 4, pp. 871–884, Apr. 2020.
[27] S. B. Akers Jr., "Synthesis of combinational logic using three-input majority gates," in *Proc. 3rd Annu. Symp. Switch. Circuit Theory Logical Design*, 1962, pp. 149–157.
[28] R. McNaughton, "Unate truth functions," *IRE Trans. Electron. Comput.*, vol. 10, no. 1, pp. 1–6, 1961.
[29] H. Owlia, P. Keshavarzi, and A. Rezai, "A novel digital logic implementation approach on nanocrossbar arrays using memristor-based multiplexers," *Microelectron. J.*, vol. 45, no. 6, pp. 597–603, 2014.
[30] A. Khan and R. Arya, "Design and energy dissipation analysis of simple QCA multiplexer for nanocomputing," *J. Supercomput.*, vol. 78, no. 6, pp. 8430–8444, 2022.
[31] C. Scholl and B. Becker, "On the generation of multiplexer circuits for pass transistor logic," in *Proc. Design, Autom. Test Europe*, 2000, pp. 372–378.
[32] M. Soeken et al., "The EPFL logic synthesis libraries," 2022, *arXiv:1805.05121*.
[33] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015, pp. 1–5.
[34] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. 22nd Int. Conf. Comput.-Aided Verification*, 2010, pp. 24–40.
[35] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 2, pp. 137–148, Jun. 1994.
[36] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *Proc. 27th Asia–South Pacific Design Autom. Conf.*, 2022, pp. 410–416.

Siang-Yun Lee received the B.Sc. degree from the Department of Electrical Engineering, National Taiwan University (NTU), Taipei, Taiwan, in 2019. She is currently pursuing the Ph.D. degree with the Integrated Systems Laboratory, EPFL, Lausanne, Switzerland, led by Prof. G. De Micheli.

In NTU, she worked with Prof. J.-H. R. Jiang on threshold logic synthesis. She is currently a maintainer of the EPFL logic synthesis library mockturtle. Her research interests include logic synthesis and design automation for emerging technologies.

Giovanni De Micheli (Life Fellow, IEEE) graduated in nuclear engineering from the Politecnico di Milano, Milan, Italy, in 1979. He received the M.S. and Ph.D. degrees in EECS from the University of California at Berkeley, Berkeley, CA, USA, in 1980 and 1983, respectively.

He is a Professor and a Director with the Integrated Systems Laboratory, EPFL, Lausanne, Switzerland. Previously, he was a Professor of Electrical Engineering with Stanford University, Stanford, CA, USA. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies.

Prof. De Micheli is the recipient of the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards. He is a member of the Scientific Advisory Board of IMEC (Leuven, B) and STMicroelectronics. He is a Fellow of ACM and AAAS, a Member of the Academia Europaea, and an International Honorary member of the American Academy of Arts and Sciences.