Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis

Heinz Riener
EPFL, Switzerland

Siang-Yun Lee
EPFL, Switzerland

Alan Mishchenko
UC Berkeley, USA

Giovanni De Micheli
EPFL, Switzerland

***Abstract*—The paper presents a novel DAG-aware Boolean rewriting algorithm for restructuring combinational logic before technology mapping. The algorithm, called *window rewriting*, repeatedly selects small parts of the logic and replaces them with more compact implementations. Window rewriting combines small-scale windowing with a fast heuristic Boolean resynthesis. The former uses sophisticated structural analysis to capture reconvergent paths in a multi-output window. The latter re-expresses the multi-output Boolean function of the window using fewer gates if possible. Experiments on the EPFL benchmarks show that window rewriting is competitive with state-of-the-art AIG rewriting in both quality and runtime, resulting in an average size reduction of 9.16% and outperforming the 4-cut rewriting command *drw* in ABC by 3.41%.**

I. Introduction

Logic optimizations play a key role in automated design flows for digital systems and are responsible for substantial area, delay, and power reductions. They are applied to a simple and technology-independent representation of the digital logic, typically automatically derived from a high-level description of the system. Modern logic optimization algorithms target multi-level logic representations such as *And-inverter graphs* (AIGs) [1], composed of two-fanin AND gates and inverters, or *Xor-and graphs* (XAGs) [2], AIGs enriched with two-fanin XOR gates.

Boolean rewriting [3] is a logic optimization methodology to greedily minimize a multi-level logic representation by iteratively selecting sub-graphs rooted at a node and replacing them with smaller pre-computed sub-graphs, while preserving the functionality of the root node. DAG-aware AIG rewriting [4] implements the Boolean rewriting methodology for AIGs. It has been made scalable by combining cut-enumeration [5], [6], fast truth table-based manipulation, and computing a canonical representation of Boolean functions [7]. For each network node v, the DAG-aware rewriting algorithm enumerates a fixed number of sub-graphs rooted at v with at most k inputs, called *k-feasible cuts*. In practise [4], due to scalability considerations, the cut size k has been fixed to 4 and up to 8 cuts are considered per node. Attempts to further improve the quality of Boolean rewriting by enumerating more and larger sub-graph structures per node, such as [8], [9], were not able to build on the success of DAG-aware AIG rewriting because they suffer from one or more of the following limitations:

- The number of k-feasible cuts per node significantly increases with k such that considering many (or all) cuts per node is often practically impossible without taking considerable runtime degradation into account.
- Mining and storing a database with optimum implementations of Boolean functions of 5 or more inputs, as well as searching over the database, become more time-consuming and require more memory. Often only the most frequently appearing functions are stored [8].
- Approaches based on exact synthesis [9], which compute optimum implementations for each new cut function on-the-fly and cache them, do not scale well due to the high runtime requirements of exact synthesis.

In this paper, we propose a new way to perform Boolean rewriting. The two main characteristics that distinguish the proposed rewriting, called *window rewriting*, from the previous approaches can be summarized as follows:

1) *Reconvergence-driven windowing*: Instead of enumerating a large number of single-output cuts per node, our approach uses sophisticated structural analysis to construct only one multi-output window per node. The window is constructed to include reconvergent paths in the vicinity of the node. The existence of such a reconvergence in a sub-graph is a necessary condition for a don't-care-based size reduction of the circuit. We argue that one 6-input window (with possibly many outputs) constructed this way contains many 4-feasible cuts and, consequently, provides the same or better optimization capabilities, compared to the classical rewriting based on cut enumeration.
2) *Heuristic Boolean resynthesis*: To optimize multi-output windows, we generalize Boolean resubstitution, a technique that attempts to re-express a node's function using other nodes already present in the network. We have developed a high-effort Boolean resynthesis engine capable of efficiently resynthesizing multi-output Boolean functions utilizing don't-cares and existing node functions. The computation performed by our engine is local and does not require any pre-computed database information. In contrast, traditional resubstitution algorithms run a trial-and-error search (with filtering) to resynthesize a target node more compactly using a fixed set of simple structures of one or two logic gates built upon some existing nodes. Our heuristic Boolean resynthesis algorithm searches for an arbitrary dependency circuit composed of potentially many logic gates to resynthesize the target

function.

Experiments on the EPFL benchmarks show that 6-input window rewriting leads to a better quality-of-results than the best implementation of AIG rewriting in ABC, `drw` [4], while being comparable in runtime. A single iteration of window rewriting improves by 3.41% over repeating `drw` until convergence. The advantage of window rewriting lies in its ability to analyze larger sub-graph structures (one 6-input window instead of many 4-feasible cuts per node) and to exploit don't-cares and existing node functions during optimization (in contrast to using a pre-computed database of optimum implementations).

The paper is structured as follows: Section II presents the background and notation. Section III details the contributions: window rewriting, reconvergence-driven windowing, and heuristic Boolean resynthesis. Section IV presents experimental results, and Section V concludes the paper.

II. Background

A. Boolean Functions

Let $\mathbb{B} = \{0,1\}$. A (single-output) *Boolean function* $f : \mathbb{B}^n \rightarrow \mathbb{B}$, $f(x) = y$, over *Boolean variables* $x = x_1, \ldots, x_n$ defines a mapping from assignments of n Boolean values to single Boolean values. A Boolean function f *depend* on variable x_i if $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \neq f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ and $\textsc{Deps}(f)$ denotes the set of all variables on which f depends.

B. Gate-Inverter Graphs

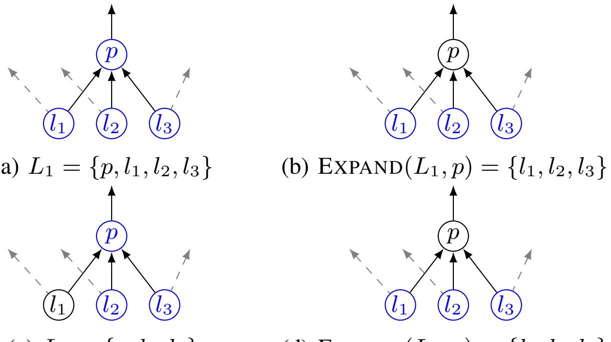
We use *gate-inverter graphs* (GIG) as a technology-independent model of combinational logic functions. A gate-inverter graph $N = (V, E, I, O)$ is a 4-tuple, where (V, E) is a directed acyclic graph with nodes V and edges $E = V^2$, and where $I \subseteq V$ is a set of primary inputs and $O \subseteq V$ is a set of primary outputs. Each node $v \in V$ models either a *primary input* or a *gate* from a predefined *gate library*. Edges, connecting nodes, model wires and can either be *regular* or *complemented*.

The *fanins* (*fanouts*) of a node $v \in V$, denoted as $\textsc{Fanins}(v)$ ($\textsc{Fanouts}(v)$), are the nodes connected to v via incoming (respectively, outgoing) edges. The k-bounded *transitive fanin-cone* $\text{TFI}_k(v)$ and k-bounded *transitive fanout-cone* $\text{TFO}_k(v)$ of a node v in N are the subsets of nodes in N reachable by traversing at most k transitive fanin-edges and at most l transitive fanout-edges starting at v. We use $\text{TFI}(v) = \text{TFI}_\infty(v)$ and $\text{TFO}(v) = \text{TFO}_\infty(v)$ to denote the unbounded transitive cones. A GIG N is *κ-regular* if all gates in N have exactly κ fanins.

Prominent examples of GIGs are *And-inverter graphs* (*And-xor graphs*), which use two-fanin AND gates (respectively, two-fanin AND and XOR gates) as a gate library. AIGs and XAGs are 2-regular.

C. Cuts and Cut Expansion

A *cut* $C = (r, L)$ in a GIG N is a pair of a node r, called *root*, and a set L of nodes, called *leaves*, such that



(a) $L_1 = \{p, l_1, l_2, l_3\}$ (b) $\textsc{Expand}(L_1, p) = \{l_1, l_2, l_3\}$

(c) $L_2 = \{p, l_2, l_3\}$ (d) $\textsc{Expand}(L_2, p) = \{l_1, l_2, l_3\}$

Fig. 1: Two examples of cost-free expansions of sets of leaves.

1) each path from any primary input of N to r passes through at least one leaf in L and
2) for each leaf $l \in L$, there is at least one path from a primary input to r passing through l and not through any other leaf.

The cover $\textsc{Cover}(C)$ of a cut $C = (r, L)$ in N is the set of nodes v in N that appear on a path from any $l \in L$ to r, without L. The *expand* operation

$$\textsc{Expand}(L, v) = \begin{cases} (L - \{v\}) \cup \textsc{Fanins}(v), & v \in L \\ L, & v \notin L \end{cases} \tag{1}$$

replaces a node v in a set L of leaves with its fanins. The *cost*

$$\Delta(L, v) = |\textsc{Expand}(L, v)| - |L| \tag{2}$$

of expanding L with a node v is the difference of the number of leaves after and before expansion. If $\Delta(L, v) \leq 0$, we call an expansion *cost-free*. It is easy to observe that $\textsc{Expand}(L, v)$ is *cost-free* iff at most one fanin of v is not in L, i.e., iff $|\textsc{Fanins}(v) - L| \leq 1$.

Two simple examples of cost-free expansions of sets of leaves are depicted in Figure 1(a)-(b) and Figure 1(c)-(d), respectively.

D. Node Function and Don't-Care Conditions

Each node v in a GIG computes a Boolean function $f_v : \mathbb{B}^n \rightarrow \mathbb{B}$, called *node function*, over variables $x_1, \ldots, x_n$, where the elementary variable $x_1, \ldots, x_n$ are assigned to the primary inputs $i_1, \ldots, i_n$. Internal flexibilities may arise in the Boolean function f_v due to limited controllability or observability at the node v. These *don't-care conditions* at the node v are modelled as a Boolean function $dc_v : \mathbb{B}^n \rightarrow \mathbb{B}$, whose value is 1 under an assignment *if and only if* (iff) the value produced by f_v does not affect the primary outputs of the GIG.

E. Reconvergence

A path p is a finite sequence $v_0, \ldots, v_l$ of nodes such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < l$. Two paths are *reconvergent* if they start at the same node v_0, end at the same node v_l, and contain, respectively, two different fanins of v_l. For the sake of simplicity, we call the corresponding node v_0 *reconvergent*.

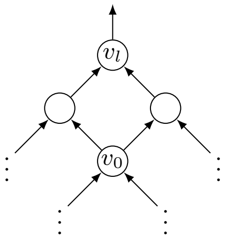


Fig. 2: Graph structure with reconvergent paths.

Figure 2 shows a pair of reconvergent paths starting at v_0 and ending in v_l.

F. Boolean Resynthesis

Optimization of a node v in a GIG can be of two types: (1) *controllability* (CDC)-based optimization that does not change the node's function f_v during optimization; and (2) *observability* (ODC)-based optimization that transforms f_v into another function $f_v^\star$, but preserves the functions of all primary outputs because the difference between f_v and $f_v^\star$ is included in the observability don't-cares of v.

Logic optimization is performed by solving *Boolean resynthesis* formulated as follows: A *target function* $f : \mathbb{B}^n \to \mathbb{B}$ is specified by its *on-set function* f_{on} and *off-set function* f_{off}. The target function can be incompletely-specified if the union of f_{on} and f_{off} does not cover the Boolean domain. Given $f_{\text{on}}, f_{\text{off}}$ and a set $\{f_{d_1}, \ldots, f_{d_r}\}$ of completely-specified *divisor functions* $f_{d_i} : \mathbb{B}^n \to \mathbb{B}$ over the same variables, find a *dependency function* $h : \mathbb{B}^r \to \mathbb{B}$, such that

$$\begin{aligned} &h\Big(f_{d_1}(x), \ldots, f_{d_r}(x)\Big) \to f_{\text{on}}(x) \text{ and} \\ &f_{\text{off}}(x) \to \neg h\Big(f_{d_1}(x), \ldots, f_{d_r}(x)\Big) \end{aligned} \tag{3}$$

for all assignments $x \in \mathbb{B}^n$.

In particular, we are interested in the *dependency circuits* that realize h with as few nodes as possible. In the context of window rewriting, the target function is the function of a selected node, called the *root* node, in the window, and the n Boolean variables it depends on are assigned to the n window inputs. The divisor functions are the functions of some other nodes, called the *divisors*, in the same window.

In this paper, we use *truth tables* to represent node functions, which are sequences of bits recording the values of the node under each combination of (local) input values and is stored in an 64-bit unsigned integer in our implementation. The number of 1-bits in the truth table of a function f is denoted as $\textsc{Ones}(f)$.

III. Window Rewriting

Boolean rewriting is a fast and greedy methodology for minimizing GIGs. **Algorithm R** summarizes the conceptual steps of eager Boolean rewriting at high level: the algorithm iteratively chooses a node as a pivot p, constructs a sub-graph in the local neighborhood of p, optimizes the logic of the sub-graph, and replace it.

Algorithm R (*Boolean Rewriting*). Given a GIG N.

R1. [Choose pivot.] Select a node p in N as pivot node.

R2. [Construct sub-graph.] Construct a sub-graph structure (I, O, G) in the local neighborhood of p with local inputs $i_1, \ldots, i_n$, local outputs $o_1, \ldots, o_m$, and inner nodes $g_1, \ldots, g_r$, where n, m, and r are the number of local inputs, local outputs, and gates.

R3. [Simulate sub-graph.] Compute the functions of all nodes of the sub-graph in topological order to obtain output function $f_{o_i} : \mathbb{B}^n \to \mathbb{B}$ for $1 \le i \le m$.

R4. [Resynthesize output functions.] Resynthesize the output functions f_{o_i}, $1 \le i \le m$, to obtain a replacement $(I, O' = \{o'_1, \ldots, o'_m\}, G')$ with functionally equivalent output functions. If $|G'| > |G|$, goto R1. Otherwise, proceed with R5.

R5. [Replace sub-graph.] Insert the gates G' in topological order into N, replace o_i with o'_i, and remove o_i from N for all $1 \le i \le m$. Goto R1. ▌

The conceptual steps can be instantiated with different strategies. In Section III-A, we propose a reconvergence-driven windowing algorithm to construct sub-graphs in **R2** that capture the reconvergent paths of a pair of nodes in form of multi-output windows. In Section III-B, we present a fast heuristic Boolean resynthesis algorithm for **R4** that resynthesizes the logic of a multi-output sub-graph using AND and XOR gates with cost-free inversions.

A. Reconvergence-Driven Windowing

In this section, we first show that reconvergence are essential for don't-care-based optimization and then introduce a reconvergence-driven window construction algorithm.

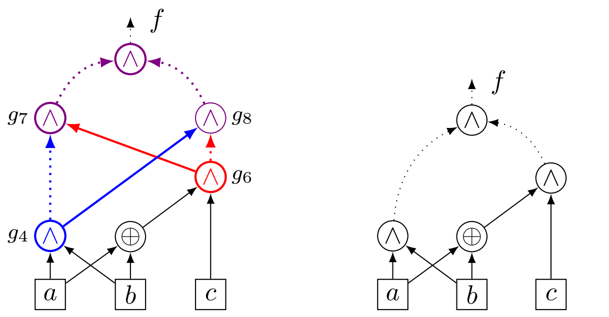
Reconvergence enables don't-care-based optimizations. We show that reconvergence is essential for don't-care-based optimizations. In our proofs, we consider controllability and observability separately.

Theorem 1. *For any node v in a κ-regular GIG N, if there exists a CDC-based optimization for v, then there must be a reconvergent node in* $\text{TFI}(v)$.

Proof of Theorem 1. We prove the reversed statement, i.e., if there is no reconvergent node in $\text{TFI}(v)$, then the cone is a minimum-size implementation of f_v. Having no reconvergent node in $\text{TFI}(v)$ means all nodes in $\text{TFI}(v)$ has only one fanout staying in $\text{TFI}(v)$. In other words, $\text{TFI}(v)$ is a *tree*. Let $\text{S}(v)$ denote the (structural) support of $\text{TFI}(v)$, which is defined as the leaves of $\text{TFI}(v)$. Since N is κ-regular, the sizes of $\text{TFI}(v)$ and $\text{S}(v)$ are related by

$$\begin{aligned} |\text{S}(v)| &= (|\text{TFI}(v)| - |\text{S}(v)|) \cdot (\kappa - 1) + 1 \\ \implies |\text{TFI}(v)| &= \frac{|\text{S}(v)| - 1}{\kappa - 1} + |\text{S}(v)| \end{aligned} \tag{4}$$

Since the functions of all nodes depend on all of their fanins, f_v depends on all nodes in $\text{S}(v)$ and none of them can be taken out. Now, we attempt to build a smaller graph N' to replace $\text{TFI}(v)$ starting from $\text{S}(v)$. At each step, we add one



(a) Original sub-graph. (b) Optimized sub-graph.

Fig. 3: Example of CDC-based optimization.

node into N' and connect κ nodes to it, which eliminate at most $\kappa - 1$ zero-fanout nodes. At least $(|\mathrm{S}(v)| - 1)/(\kappa - 1)$ steps are needed to make N' have only one zero-fanout node. A lower bound on $|N'|$ is hence derived as

$$|N'| \geq |\mathrm{S}(v)| + \frac{|\mathrm{S}(v)| - 1}{\kappa - 1} \tag{5}$$

Thus, $|\mathrm{TFI}(v)|$ is minimal. □

An example of a CDC-based optimization is illustrated in Figure 3, where a, b, c denote primary inputs, $\wedge$ denotes an AND gate, $\oplus$ denotes an XOR gate, and dotted edges denote complementation. The assignment $f_{g_4} = 1$ and $f_{g_6} = 1$ is a controllability don't-care for the XOR function implemented by the AND gates g_7, g_8, and g_9. Consequently, the three AND gates can be replaced by a single AND gate (implementing an OR function) without affecting the output function f. There is one pair of reconvergent paths ending at f and starting from g_4 and g_6, respectively.

Theorem 2. *For any node v in a κ-regular GIG N, if there exists an ODC-based optimization for v, then v must be on a reconvergent path.*

Proof of Theorem 2. Suppose the ODC-based optimization for v transforms the function of v into f_v^*. The necessary condition to preserve primary output functions is

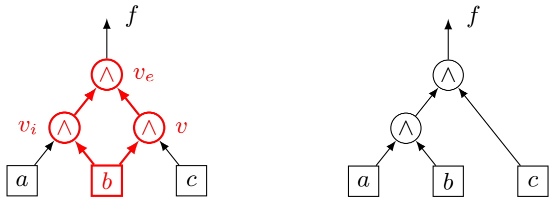
$$(f_v \oplus f_v^*) \to dc_v \implies \textsc{Deps}(dc_v) \cap \textsc{Deps}(f_v) \neq \emptyset. \tag{6}$$

Following the computation of ODCs [10], it can be shown that

$$\begin{aligned} &\exists v_e \in \mathrm{TFO}(v), v_i \in \textsc{Fanins}(v_e) : v_i \notin \mathrm{TFO}(v) \\ &\text{such that } \textsc{Deps}(f_{v_i}) \cap \textsc{Deps}(f_v) \neq \emptyset. \end{aligned} \tag{7}$$

Hence, starting from a common primary input of $\textsc{Deps}(f_v)$ and $\textsc{Deps}(f_{v_i})$, there is at least a pair of reconvergent paths ending in v_e, one passing through v and the other passing through v_i. □

To illustrate the proof of Theorem 2, an example of ODC-based optimization is shown in Figure 4, where a, b, c are primary inputs and other nodes are AND gates. An ODC-based optimization for v transforms its function from $f_v = b \wedge c$ to $f_v^* = c$. This preserves the output function $f = a \wedge b \wedge c$ because $dc_v = \neg(a \wedge b)$ and $(f_v \oplus f_v^*) \to dc_v$ holds. The reconvergent paths are colored in red.



(a) Original sub-graph. (b) Optimized sub-graph.

Fig. 4: Example of ODC-based optimization.

Theorems 1 and 2 show that the existence of reconvergence is a necessary condition for both CDC-based and ODC-based optimization. This motivates us to develop a reconvergence-driven windowing algorithm which prioritizes the inclusion of reconvergent paths.

Remark. Although reconvergence is necessary for optimizing a single node, when the size of a multi-output GIG is considered, another type of optimization is possible without the existence of reconvergence, namely *logic sharing*. It is possible that the TFIs of all primary outputs are of minimum size, but there exist several different implementations of one output that can be partly shared with the TFIs of other outputs to further decrease the overall size of the GIG. For example, for a GIG with two primary outputs f, g and three primary inputs x, y, z, the following sub-optimal implementation has no reconvergence:

$$f = (x \wedge y) \wedge z, \;\; g = x \vee (y \wedge z) \tag{8}$$

The optimal GIG with logic sharing of $y \wedge z$ is:

$$f = x \wedge (y \wedge z), \;\; g = x \vee (y \wedge z) \tag{9}$$

Window construction. For a given pivot node p, **Algorithm W** identifies another node m such that there exists a pair of reconvergent paths between p and m and m has shortest distance to p. The algorithm then collects the nodes on the two reconvergent pathts and expands the sub-graph structure towards the input and output boundary.

Algorithm W (*Window construction*). Given a node p, called *pivot*, in a GIG N and two integers k and l, called *cut size* and *distance*, this algorithm computes a small-scale window in the local neighborhood of p with at most k inputs and potentially multiple outputs that reaches a reconvergence in at most l steps. The window is characterized by a triple (I, O, G), of local inputs I, local outputs O, and inner nodes G.

W1. [*Identify and collect reconvergence.*] Use breadth-first search to identify a node m, called *meet*, reachable from two fanins of p in at most l steps and add all nodes on the two paths from p to m to G.

W2. [*Identify and collect inputs.*] Mark all nodes in G as visited. Iterate over all fanins of the nodes in G and add all nodes without marks to I. If $|I| > k$, terminate without returning a result.

W3. [*Expand towards TFI.*] As long as $|I| \leq k$ and not all nodes in I are primary inputs, repeat two steps: (a) first perform all cost-free expansions of I; (b) then choose a fanin v from I such that $|\textsc{Expand}(I, v)| \leq k$ and v has a highest fanout count within G. Expand I with n and go to (a). Otherwise, if no further expansion is possible in (b), go to the next step.

W4. [*Expand towards TFO.*] Define a (sorted) map L which assigns a (initially empty) set of nodes to each level of N. Iterate over all nodes in $I \cup G$, mark them, and sort them into L at the corresponding levels. Iterate over the nodes at each level in L from lowest to highest level. For each node, systematically explore its fanouts. If a fanout v is an inner node and not marked, but all its fanins are marked, then add v to G, mark it, and sort it into L.

W5. [*Identify and collect window outputs.*] Define a zero-initialized reference counter for each node in N. Iterate over all nodes in G and increment the counters of all fanins of $g \in G$. Iterate again over all nodes in G, observe their reference counters, and mark a node as output if its reference counter is lower than its fanout count.

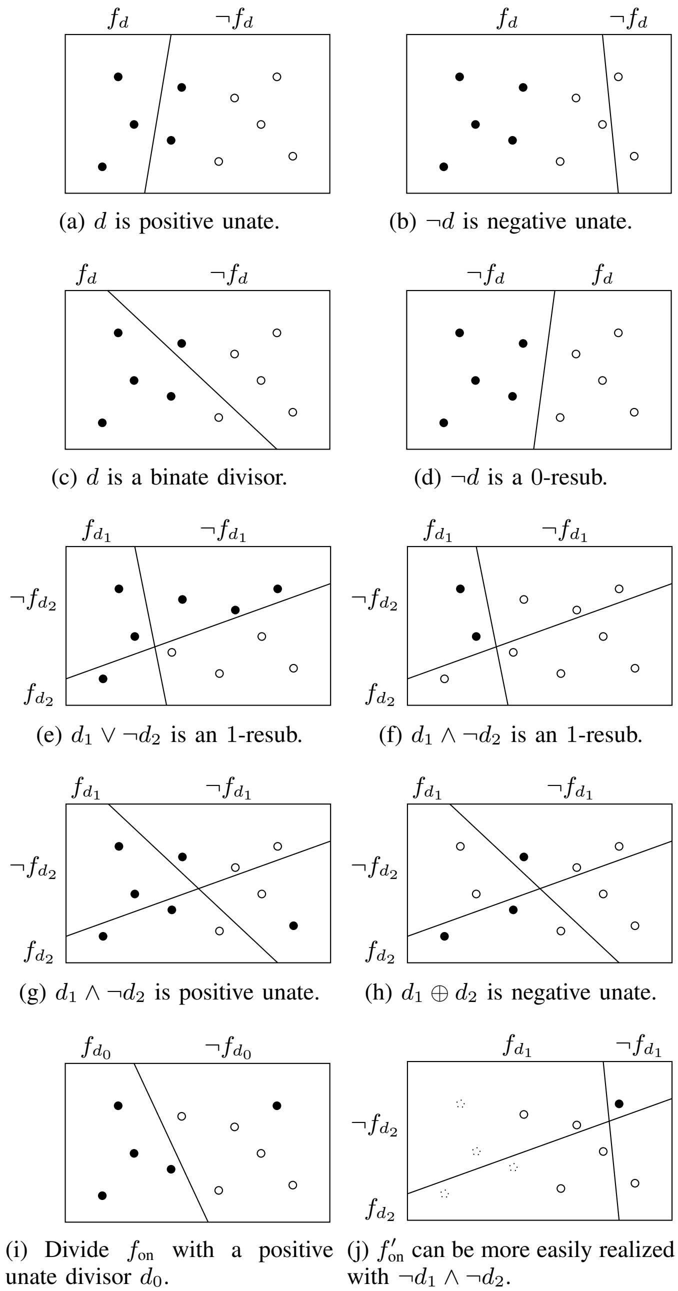
W6. [*Topologically sort.*] Sort the nodes in G in topological order with respect to the structure of N and return the triple (I, O, G). ▌

B. Heuristic Boolean Resynthesis

In this section, we propose a heuristic algorithm to solve the logic resynthesis problem using AND and optionally XOR gates with cost-free inversions. The algorithm is based on a recursive decomposition that classifies divisors on their intersections with the on-set and the off-set of a target function.

Given the target on-set function f_{on} and off-set functions f_{off}, a divisor d (or its negation $\neg d$) is said to be *positive unate* if $f_d \wedge f_{\text{off}} = 0$ (or if $\neg f_d \wedge f_{\text{off}} = 0$). Similarly, a divisor d is said to be *negative unate* if $f_d \wedge f_{\text{on}} = 0$. If both d and $\neg d$ are neither positive nor negative unate, then d is said to be a *binate* divisor. For example, in Fig. 5 (a), d is positive unate because $f_d \wedge f_{\text{off}} = 0$; in Fig. 5 (b), $\neg d$ is negative unate because $\neg f_d \wedge f_{\text{on}} = 0$; and in Fig. 5 (c), d is a binate divisor because neither d nor $\neg d$ is unate. As the unateness property is independent for d and $\neg d$, from now on, a divisor with optional negation is referred to as a *literal*, i.e., a literal is either a divisor or a negated divisor.

If a literal l is positive unate and its negation $\neg l$ is negative unate, then l realizes the target. We call this a *0-resub* because it is a resubstitution with 0 additional nodes. For example, in Fig. 5 (d), $\neg d$ is a 0-resub because $\neg d$ is positive unate and d is negative unate. Two positive unate literals may be combined with an OR gate (implemented in AIGs and XAGs as an AND gate with input and output negations) to obtain a larger intersection with the on-set. If all on-set minterms are contained in the union of two literals l_1, l_2, then $l_1 \vee l_2$ realizes the target and we call it an *OR-type 1-resub*. To find such cases, we check if $\neg(f_{l_1} \vee f_{l_2}) \wedge f_{\text{on}} = 0$, i.e., if $\neg f_{l_1} \wedge \neg f_{l_2} \wedge f_{\text{on}} = 0$. For example, in Fig. 5 (e), d_1 and $\neg d_2$ are two positive unate literals and $d_1 \vee \neg d_2$ is an OR-type 1-resub because $\neg f_{d_1} \wedge f_{d_2} \wedge f_{\text{on}} = 0$. Similarly, two negative unate literals l_1, l_2 can be combined with an AND gate to form an *AND-type 1-resub* if $\neg f_{l_1} \wedge \neg f_{l_2} \wedge f_{\text{off}} = 0$. For example, in Fig. 5 (f), $\neg d_1$ and d_2 are two negative unate literals and $d_1 \wedge \neg d_2$ is an AND-type 1-resub because $f_{d_1} \wedge \neg f_{d_2} \wedge f_{\text{off}} = 0$.



(a) d is positive unate.

(b) $\neg d$ is negative unate.

(c) d is a binate divisor.

(d) $\neg d$ is a 0-resub.

(e) $d_1 \vee \neg d_2$ is an 1-resub.

(f) $d_1 \wedge \neg d_2$ is an 1-resub.

(g) $d_1 \wedge \neg d_2$ is positive unate.

(h) $d_1 \oplus d_2$ is negative unate.

(i) Divide f_{on} with a positive unate divisor d_0.

(j) f'_{on} can be more easily realized with $\neg d_1 \wedge \neg d_2$.

Fig. 5: Illustration of **Algorithm S**. Black dots represent on-set minterms, white dots represent off-set minterms, and dotted circles represent don't-care minterms.

The definitions of positive and negative unateness are extended for pairs of literals. A pair p of two literals l_1, l_2

obtained from binate divisors can be combined using an AND gate (or an XOR gate, if XAGs are used) to construct a new function $f_p = f_{l_1} \wedge f_{l_2}$ (or $f_p = f_{l_1} \oplus f_{l_2}$ if XOR is used). The pair is said to be positive unate if $f_p \wedge f_{\text{off}} = 0$, and is said to be negative unate if $f_p \wedge f_{\text{on}} = 0$. For example, in Fig. 5 (g), d_1 and d_2 are both binate divisors and $(d_1, \neg d_2)$ form an AND-type positive unate pair because $f_{d_1} \wedge \neg f_{d_2} \wedge f_{\text{off}} = 0$; in Fig. 5 (h), d_1 and d_2 are both binate divisors and (d_1, d_2) form an XOR-type negative unate pair because $(f_{d_1} \oplus f_{d_2}) \wedge f_{\text{on}} = 0$.

Algorithm S (*Heuristic Boolean Resynthesis*). The inputs to this algorithm are the target on-set and off-set functions $f_{\text{on}}, f_{\text{off}}$ and a set $D = \{d_1, \dots, d_r\}$ of divisors associated with divisor functions $f_{d_1}, \dots, f_{d_r}$.

S1. [*Constants.*] Check if $f_{\text{on}} = 0$ or if $f_{\text{off}} = 0$. If so, return the constant 0 or 1.

S2. [*Classify divisors.*] For each divisor d and its negation $\neg d$, check if they are positive or negative unate. If both of them are not unate, classify d as binate.

S3. [0-*resub.*] Check the collected lists of positive and negative unate literals for an 0-resub and return it if found.

S4. [*Sort unate literals.*] Sort the positive unate literals by the number of on-set minterms $\textsc{Ones}(f_l \wedge f_{\text{on}})$, and sort the negative unate literals by the number of off-set minterms $\textsc{Ones}(f_l \wedge f_{\text{off}})$.

S5. [1-*resub.*] Enumerate pairs (l_1, l_2) of positive unate literals to find an OR-type 1-resub. With the order sorted in S4, the enumeration can be terminated earlier if we know

$$\textsc{Ones}(f_{l_1} \wedge f_{\text{on}}) + \textsc{Ones}(f_{l_2} \wedge f_{\text{on}}) < \textsc{Ones}(f_{\text{on}}) \quad (10)$$

for the rest of the list. Similarly, enumerate pairs of negative unate literals to find an AND-type 1-resub and return it if found.

S6. [*Collect and sort unate pairs.*] For each pair of binate divisors d_1, d_2, test the unateness of combining them using an AND gate and with all the four possibilities of negations. If XOR gates are allowed, test also the unateness of combining them using an XOR gate or an XNOR gate. Collect the positive and negative unate pairs and sort them using the same method as in S4.

S7. [2- *and* 3-*resub*] Similar to S5, try to find a 2-resub by combining a unate literal and a unate pair. Then, try to find a 3-resub by combining two unate pairs.

S8. [*Recursive construction.*] When the target cannot be realized within 3 gates, the algorithm heuristically choose an unate literal or an unate pair to decompose the function. If a positive unate literal (or pair) l_p is chosen, a new on-set function $f'_{\text{on}} = f_{\text{on}} \wedge \neg f_{l_p}$ with fewer minterms is derived using an OR gate on top of the dependency circuit and having l_p as one of the fanins of the OR gate. Then, **Algorithm S** is recursively called on the new f'_{on} to construct the remaining circuit at the other fanin of the OR gate. For example, in Fig. 5 (i), the original target $(f_{\text{on}}, f_{\text{off}})$ and the function of a positive unate literal d_0 is shown. Dividing f_{on} with f_{d_0}, the new $f'_{\text{on}} = f_{\text{on}} \wedge \neg f_{d_0}$ is shown in Fig. 5 (j). Then, the new target $(f'_{\text{on}}, f_{\text{off}})$ is realized by $\neg d_1 \wedge \neg d_2$, resulting in the final solution $d_0 \vee (\neg d_1 \wedge \neg d_2)$. Similarly, if a negative unate literal (or pair) l_n is chosen, an AND gate is used on top of the dependency circuit and a new off-set function $f'_{\text{off}} = f_{\text{off}} \wedge \neg f_{l_n}$ is derived. ▮

C. Multi-output Window Optimization

In the previous sections, **Algorithm W** constructs a multi-output window and **Algorithm S** can be used to resynthesize the function of a node using the functions of some other nodes. To optimize the multi-output window, in this section, **Algorithm M** runs **Algorithm S** on each node in a window, trying to optimize its local implementation using the don't-care conditions computed within the window.

Algorithm M (*Multi-output Resubstitution*). Given a multi-output window (I, O, G) with input nodes I, output nodes O, and inner nodes G, this algorithm resynthesizes the output functions f_{o_i}, $1 \leq i \leq m$ to obtain a replacement (I, O', G') with functionally equivalent output functions.

M1. [*Initialize.*] Initialize G' with G. Let $T = \emptyset$ be the set of already-tried nodes. Associate each input nodes $i_j \in I$ with the j^{th} projection function. Simulate the window in a topological order to obtain the functions of all inner nodes and output nodes in terms of the input nodes.

M2. [*Choose the node to resynthesize.*] Select a node $r \in G, r \notin T$ in a reversed topological order.

M3. [*Compute ODC.*] Temporarily complement the function of r, i.e., let $f'_r = \neg f_r$, and re-simulate $\text{TFO}(r)$ to obtain f'_{o_i} for each output node o_i. Compute the ODC:

$$dc_r = \neg \bigvee_{o_i \in O} f_{o_i} \oplus f'_{o_i}. \quad (11)$$

M4. [*Mark MFFC.*] Mark the nodes in the *maximum fanout-free cone* (MFFC) [6] of r. A node v is in the MFFC of r if $v \in \text{TFI}(r)$ and all paths from v to any output node pass through r.

M5. [*Collect divisors.*] Collect the divisors $D = I \cup G' - \text{MFFC}(r) - \text{TFO}(r)$.

M6. [*Resubstitute node.*] Resynthesize f_r using the collected divisors D by calling **Algorithm S** with

$$f_{\text{on}} = f_r \wedge \neg dc_r \quad \text{and} \quad f_{\text{off}} = \neg f_r \wedge \neg dc_r. \quad (12)$$

If a dependency circuit $(I_S \subseteq D, O_S = \{r'\}, G_S)$ is resynthesized and $|G_S| < |\text{MFFC}(r)|$, update G' with

$$G' \cup G_S \cup \{r'\} - \text{MFFC}(r) - \{r\} \quad (13)$$

Add r or r' into the set of tried nodes T. Goto M2. ▮

TABLE I: Statistics of windowing on EPFL benchmarks.

Property	Total	Contained	
Node containment	467399	458260	98.04%
4-Cut containment	4770189	1949063	40.86%

TABLE II: Heuristic synthesis statistics for completely-specified 3-input and 4-input Boolean functions.

		Heuristic Resynthesis				Exact Database	
Repr	Var	Success	Failed	ANDs	XORs	ANDs	XORs
AIG	3	254	2	890	0	794	0
	4	54622	10914	499308	0	365276	0
XAG	3	254	2	528	142	384	206
	4	54622	10914	351592	60332	178536	98940

IV. Experimental Evaluation

Window rewriting has been implemented in C++ and experiments have been conducted using the EPFL benchmark suite. In the Sections IV-A and IV-B, the performance of **Algorithm W** (Window construction) and **Algorithm S** (Heuristic Boolean Resynthesis) are analysed. In Section IV-C, a comparison between 4-cut rewriting and 6-input window rewriting is presented.

A. Quality of Reconvergence-Driven Windowing

In this section, we investigate the quality of the reconvergence-driven windowing algorithm (proposed in Section III-A) by structurally analyzing the windows constructed for the EPFL benchmark suite. We consider each node in the EPFL benchmarks as a pivot node and run **Algorithm W** and a conventional 4-cut enumeration algorithm to test node containment (how many nodes are contained at least once in a window) and cut containment (how many 4-feasible cuts are completely contained in a window). We say that a cut is *contained* iff its cover is a subset of the window nodes. Table I summarizes our results. The table lists the total number of nodes and 4-feasible cuts (Total) generated for all benchmarks and the number of nodes and 4-feasible cuts contained in a window (Contained), respectively. The 6-input windows produced by the algorithm contain 98.04% of all nodes at least once and 40.86% of all 4-feasible cuts. One node, on average, contributes to 6.39 6-input windows.

B. Quality of Heuristic Boolean Resynthesis

In this section, we analyse the quality of the heuristic Boolean resynthesis algorithm (proposed in Section III-B) experimentally considering all completely-specified 3-input and 4-input Boolean functions. As a baseline for the comparison, we use an exact database of all NPN-4 functions containing size-minimum AIG and XAG implementations.

Table II summarizes the synthesis statistics. The table is structured as follows: the first two columns list the logic representation (Repr) and the number of variables (Var). The next four columns present the number of times the heuristic algorithm is capable to derive a logic implementation for the function (Success) and the number of times the algorithm fails to do so (Failed), and the total number of gates (ANDs, XORs) of the obtained logic implementations. The last two columns present the numbers of gates (ANDs, XORs) for the successfully synthesized functions found in a database of minimum-node implementations. Heuristic resynthesis succeeds for 99.22% and 83.55% of all 3-input and 4-input Boolean functions, respectively. The two 3-input functions heuristic resynthesis fails to synthesize are XOR3 and its complement, whose possible AIG and XAG structures all have an XOR2 component on top (either with an XOR gate or with three AND gates). In **Algorithm S**, XOR gates are only used in **S6** to construct unate pairs with binate divisors and are not considered as the top gate in **S7** or **S8**, thus it is impossible to construct a circuit with a topmost XOR gate. An XOR2 function realized with three AND gates is not possible either, because the recursive construction in **S8** builds only tree-like circuits with disjoint sub-graphs at the two fanins of the topmost gate. The average size-overhead of an implementation derived by the proposed method over the minimum-size implementation accounts for 0.35 and 2.05 gates per 3- and 4-input function, respectively.

C. Comparison with 4-Cut AIG Rewriting

All experiments targeting AIG rewriting have been conducted on a 3.5 GHz Intel Core i7 CPU with 16 GB RAM. The resulting AIGs have been verified after rewriting using the combinational equivalence checker (`&cec`) of ABC [11].

Table III shows a comparison of window rewriting and 4-cut rewriting (`drw`) using the EPFL benchmark suite. The initial benchmarks have been pre-processed using SAT sweeping [12] (`&fraig -x -C 50000`) to merge gates that are proven equivalent modulo complementation. The table is structured as follows: the first three columns name the benchmarks and present their sizes and depths after pre-processing measured in AND gates. The remaining columns list the synthesis results for each benchmark after one iteration of `drw` and window rewriting and after repeating them until convergence. Window rewriting achieves an average size reduction of 8.86% in one iteration, which increases to 9.16% when applied repeatedly. When compared to 4-cut rewriting, one iteration of window rewriting improves by 3.61% and 3.41%, respectively.

V. Conclusion

This paper presents an unique attempt to enhance Boolean circuit rewriting. The proposed algorithm builds on (1) sophisticated structural analysis to identify and capture pairs of nodes with reconvergent paths in multi-output windows and (2) fast heuristic Boolean resynthesis to optimize the logic in a multi-output window with 6 or more inputs utilizing additional divisor functions and don't-care-based optimization.

The proposed technique addresses the two exponential bottlenecks for scaling-up Boolean rewriting beyond the capabilities of 4-cut rewriting. The windowing is driven by reconvergences in the circuit structure, avoiding enumerating and prioritizing cuts whose number grows exponentially in

TABLE III: Comparison of 4-cut rewriting and 6-input window rewriting.

Benchmark			**ABC drw**							**Window rewriting**						
			First iteration			**Until convergence**				**First iteration**			**Until convergence**			
Name	Size	Depth	Size	Depth	Time	Size	Depth	Iter	Time	Size	Depth	Time	Size	Depth	Iter	Time
adder	1020	255	1020	255	0.07	1020	255	1	0.07	892	256	0.01	892	256	2	0.02
bar	3336	12	3141	12	0.09	3141	12	2	0.18	3124	12	0.10	2952	12	17	1.55
div	29040	4374	20952	43724	0.48	20833	4350	4	1.45	20985	4350	1.13	20907	4352	4	2.86
hyp	214306	24800	213118	24800	3.40	213108	24800	5	16.76	205006	24804	5.73	205004	24804	3	15.62
max	2865	287	2862	287	0.09	2862	289	2	0.18	2798	321	0.04	2765	367	3	0.12
sin	5353	222	5124	222	0.13	5108	219	4	0.52	5089	218	0.35	5089	218	2	0.63
sqrt	24506	5057	18379	5057	0.74	18371	6048	3	1.42	18265	7277	1.34	18265	7277	2	1.96
square	18482	251	17754	251	0.29	17629	249	7	1.91	16971	251	0.44	16963	251	3	1.55
arbiter	11839	87	11839	87	0.19	11839	87	1	0.19	11839	87	0.22	11839	87	1	0.22
cavlc	690	16	681	16	0.06	680	16	3	0.20	620	19	0.14	614	19	4	1.14
ctrl	169	10	125	10	0.06	122	9	3	0.18	91	17	0.01	89	17	3	0.03
dec	304	3	304	3	0.07	304	3	1	0.07	304	3	0.00	304	3	1	0.02
i2c	1321	20	1265	20	0.07	1264	19	3	0.21	1262	21	0.01	1262	21	2	0.02
int2float	258	16	224	16	0.06	221	16	4	0.25	222	18	0.01	222	18	2	0.02
mem_ctrl	46717	115	46115	115	0.56	46069	117	5	2.88	44008	122	0.95	43318	121	7	6.37
priority	978	250	852	250	0.07	695	250	19	1.32	576	65	0.01	521	47	19	0.07
router	257	54	246	54	0.06	246	52	2	0.12	173	49	0.01	160	45	3	0.01
voter	11925	65	9042	65	0.21	8899	60	3	0.53	8079	62	0.19	7993	61	3	0.63
Total	373366		353043		6.70	352411			28.44	340304		10.69	339159			32.84
Improv.			5.44%			5.61%				8.86%			9.16%			

the cut size. A fast Boolean resynthesis engine allows us to compute replacements online, and avoids the pre-generation of a database of optimised circuit structures. Pre-generating the database is a hurdle due to the exponential growth of the number of Boolean functions. Storing a database for all 6-variable Boolean functions needs an enormous amount of memory and does not allow to consider multiple outputs or logic sharing of existing divisors in the current circuit structure. Optimizing the size of the database by storing only a subset of circuit structures for frequently-occurring Boolean functions tends to be a tedious manual task and leads to a biased rewriting algorithm that is only effective for a fixed set of benchmarks. By eliminating the dependency of rewriting on a pre-computed database, the technique keeps all computations local and has best preconditions for parallelisation—there is no need for sharing a huge database with slow access times between writing workers.

In an experimental comparison between our prototypical implementation of window rewriting and all AIG rewriting algorithms in ABC (`rewrite`, `irw`, `drw`[1]) using the EPFL benchmark suite, a single iteration of window rewriting produces better results than running any AIG rewriting algorithm until convergence. Our current implementation of rewriting considers the number of AIG nodes as the objective function. An implementation focusing on size optimization of majority-inverter graphs has been presented by Lee et al. [13]. Other objective functions are possible, but require novel heuristics to keep the algorithm fast and practical. Recent experiments with simulation-guided Boolean resubstitution [14] show that the proposed resynthesis engine can be also integrated with other optimization frameworks.

[1] We have only reported the numbers for `drw`, the latest and best implementation of AIG rewriting in ABC.

References

[1] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE TCAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[2] I. Hálecek, P. Fiser, and J. Schmidt, "Are XORs in logic synthesis really necessary?," in *DDECS 2017*, pp. 134–139, 2017.

[3] P. Bjesse and A. Borälv, "DAG-aware circuit compression for formal verification," in *ICCAD 2004*, pp. 42–49, 2004.

[4] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC 2006*, pp. 532–535, 2006.

[5] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE TVLSI*, vol. 2, no. 2, pp. 137–148, 1994.

[6] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *FPGA 1999*, pp. 29–35, 1999.

[7] M. A. Harrison, "The number of equivalence classes of boolean functions under groups containing negation," *IEEE Trans. Electron. Comput.*, vol. 12, no. 5, pp. 559–561, 1963.

[8] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *ICCD 2011*, pp. 429–430, 2011.

[9] 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 *DATE 2019*, pp. 1649–1654, 2019.

[10] M. Damiani and G. De Micheli, "Observability don't care sets and Boolean relations.," in *ICCAD*, pp. 502–505, 1990.

[11] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV 2010*, pp. 24–40, 2010.

[12] L. G. Amarù, F. S. Marranghello, E. Testa, C. Casares, V. N. Possani, J. Luo, P. Vuillod, A. Mishchenko, and G. D. Micheli, "SAT-sweeping enhanced for logic synthesis," in *DAC 2020*, pp. 1–6, 2020.

[13] S. Lee, H. Riener, and G. D. Micheli, "Logic resynthesis of majority-based circuits by top-down decomposition," in *DDECS 2021*, pp. 105–110, IEEE, 2021.

[14] S. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "Simulation-guided Boolean resubstitution," 2020. arXiv:2007.02579.