Cost-generic Resubstitution lgorithm with Customizable Cost Functions

Hanyu Wang
ETH, Zurich, Swizterland
hanywang@student.ethz.ch

Siang-Yun Lee
EPFL, Lausanne, Swizterland
siang-yun.lee@epfl.ch

Giovanni De Micheli
EPFL, Lausanne, Swizterland
giovanni.demicheli@epfl.ch

***bstract*—Logic synthesis algorithms have been developed for decades optimizing technology-independent representations of digital circuits according to cost metrics defined for CMOS. However, as new technologies and applications emerge, these cost estimations correlate less to the actual costs of the products. Specialized algorithms can be developed to achieve a better quality of result (QoR), but they target only some specific applications and are less effective for others. In this work, we develop a cost-generic optimization framework adaptive to a wide range of customizable cost functions. The framework is based on Boolean resubstitution and chooses optimization candidates greedily according to the given cost function. The implementation is open-source and enables fast experimentation and prototyping of newly-defined cost estimations before developing specialized algorithms. We demonstrate with experimental results that our framework achieves comparable QoR to specialized algorithms.**

I. Introduction

Since the birth of *Electronic Design utomation* (ED) as an industry in the 1980s, design flows and algorithms have been developed mainly targeting CMOS integrated circuits. To address the NP-hard problems efficiently, the stage of logic synthesis has been divided into technology-independent optimization and technology mapping [1]. In the former stage, algorithms work on a technology-independent representation of the logic-gate-level netlist, such as the *ND-Inverter Graph* (IG), whose simpler data structure allows for a more efficient algorithm design [2]. Cost metrics based on IG properties (e.g., number of gates and logic levels), which logic synthesis algorithms optimize for, usually serve as reasonable estimations of the actual costs of the final product (area and delay, respectively) because the N ND-based CMOS libraries relate closely to ND gates in IGs [3].

However, IG-based cost metrics may not be good estimations for different target technologies or applications. For example, highly-optimized IGs do not always result in smaller *Look-Up Table* (LUT) networks after LUT mapping for FPG s. [4] s another example, for cryptography and security applications, XOR gates are preferred over ND gates [5], and in quantum circuits, XOR gates are much cheaper than ND gates [6], but an XOR gate can only be represented as three ND gates in an IG. Thus, some specialized algorithms have been developed targeting more accurate cost metrics for specific applications. For example, *Multiplicative Complexity* (MC) [7] optimization algorithms minimize the number or depth of ND gates while treating XOR gates as cost-free [8].

Nowadays, as the available computation power has dramatically grown since the first ED tools were developed, early consideration of more accurate cost functions is no longer impossible. The success of specialized optimization algorithms shows the importance of adopting better-estimated cost metrics. However, these algorithms can only be applied effectively on a limited set of targets. s beyond-CMOS technologies emerge, more complicated relations of the technology-independent representations and the actual costs may come in place. Moreover, approximation is inevitable. Gaps in cost evaluations come from not only the difference between technology-independent representations and actual products, but also propagation errors between the global cost of the entire circuit and the local cost estimation used by optimization algorithms to make decisions. When faced with various emerging technologies and possibly even more potential cost functions, a framework to quickly prototype and evaluate them is in need.

To this end, we propose a cost-generic optimization framework where cost functions are highly customizable. The framework is based on Boolean resubstitution, which computes optimization candidates locally and on the fly. In this paper, we first formalize a recursive definition of customizable cost functions, which is adaptable to a wide range of practically-useful costs. Then, we propose a cost-generic resynthesis algorithm to find the minimum-cost optimization candidate. We demonstrate that the proposed framework is capable of optimizing various cost functions. Our first experiment shows that greedily optimizing for a local cost estimation does not always lead to the best global cost evaluation. This suggests that different cost functions should be experimented and compared before developing specialized algorithms. Moreover, when the cost function is chosen well, experimental results show that our framework achieves a comparable quality of results (QoR) compared to the state-of-the-art specialized algorithm.

II. B ckground

. *Logic Functions and Resynthesis*

(n-input, single output) *Boolean function* is a function $f(\vec{x}) : \mathbb{B}^n \to \mathbb{B}$ over n variables $\vec{x} = x_1, \ldots, x_n$ in the *Boolean space* $\mathbb{B} = \{0, 1\}$.

Logic resynthesis (or simply *resynthesis*) is the problem of re-expressing a Boolean function in terms of other Boolean functions [9]: Given a target function $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = x_1, \ldots, x_k$ and a collection $G = \{g_1, \ldots, g_n\}$ of n existing functions $g_i : \mathbb{B}^k \to \mathbb{B}$ over the same variables, find a *dependency function* $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})). \quad (1)$$

for all $\vec{x} \in \mathbb{B}^k$. h does not necessarily depend on all of its n inputs. We refer to the circuit realization of h as the *dependency circuit*.

B. Logic Networks

Logic networks, or simply *networks*, are technology-independent representations of gate-level digital circuits. network is a directed acyclic graph whose vertices, referred to as *nodes*, represent logic gates or *primary inputs* (PIs), and edges represent wires. Incoming edges of a node n are referred to as the *fanins* of n and the set of fanins is represented as δ (n). Similarly, outgoing edges of a n are referred to as the *fanouts* of n. Examples of logic networks include *ND-Inverter Graphs* (IGs) [2], where every node represents a 2-input ND gate and edges can be complemented to represent an inverter, and *XOR- ND-Inverter Graphs* (X Gs) [10], where nodes represent 2-input ND or XOR gates and edges can also be complemented.

cut C in a network is a tuple (r, L) of a *root* node r and a set of *leaf* nodes L, such that every path from a PI to r passes through a leaf in L. node n is said to be *supported* by a cut $C = (r, L)$ if n fulfills the condition of being the root of C. k-feasible cut is a cut $C = (r, L)$ having no more than k leaves, i.e., $|L| \leq k$. ll k-feasible cuts in a network can be enumerated by visiting each node in a topological order and composing each pair of cuts rooted at its fanins [11]. Moreover, a higher-quality cut, a *reconvergence-driven cut*, rooted at a node r can be computed by heuristically picking one leaf n to be replaced by its fanins (to *expand* on n) [12].

cone is the set of nodes on any path between a node n and any leaf node in a cut rooted at n. The *transitive-fanin cone* (TFI) of a node n is the cone between n and the set of PIs. *fanout-free cone* (FFC) of a node n is a cone between n and a cut C, where all paths from any leaf in C to any PO pass through n. The *maximum fanout-free cone* (MFFC) of a node n is the maximum-sized FFC of n. In other words, none of the nodes in the MFFC of n has fanouts outside of the cone, such that if n is removed from the network, the MFFC can also be removed. The MFFC of a node can be identified by recursively dereferencing and referencing the TFI of n [3].

C. Technology-Independent Logic Optimization

Logic optimization algorithms can be classified into *algebraic methods* and *Boolean methods*. lgebraic methods, as its name suggests, try to apply algebraic rules based on the local structures in the network to perform small-scale restructuring [13], [14]. In contrast, Boolean methods compute Boolean functions in the network and find the different realization of the sub-networks, possibly leveraging don't-care information [3], [12], [15], [16]. Boolean methods generally result in more significant restructuring and achieve better performance than algebraic methods due to their ability to exploit Boolean logic and don't-cares. Thus, most state-of-the-art optimization algorithms are Boolean methods. Despite appearing in the literature under various names, most Boolean methods adopt either one of the two basic underlying algorithms, *rewriting* or *resubstitution*, when it comes to restructuring and optimizing a given network.

Rewriting algorithms [17] generally include the following steps:

1) Enumerate cuts for each node n in the network.
2) Simulate from each cut to get the local function of n.
3) Find optimal implementations for the computed functions by *exact synthesis* [15] or looking up in a pre-computed database [3].
4) Evaluate each candidate by dry-replacing n with each subgraph.
5) Rewrite the network using the chosen candidates.

Resubstitution algorithms [12] include the following steps:

1) For each node n in the network, compute a reconvergence-driven cut C.
2) Construct a *window* using the cut, which includes the cone between n and C and nodes outside of the cone but supported by C.
3) Simulate the window.
4) Resynthesize n by solving the resynthesis problem using the local function of n as the target and the local functions of nodes in the window but not in the MFFC of n as divisors.
5) Replace n with the resynthesized dependency circuit.

The key difference between rewriting and resubstitution is the use of *divisors*. In rewriting, substitution candidates are subgraphs implementing the target local function using the cut leaves, and logic sharing with existing nodes is identified in the evaluation step by dry-run replacement. In contrast, resubstitution identifies the divisor nodes, which are outside of the MFFC and will not be removed after substituting n, and use them as stepping stones to resynthesize n.

III. Cost Function Definition

The ultimate goal of logic optimization is to reduce the *cost*, such as area and delay, of the final chip product. However, as there are many ED processes between technology-independent logic synthesis and tape-out, such as technology mapping, placement, and routing, it is challenging to compute the actual product cost at this stage, and thus estimation is inevitable. In practice, logic optimization algorithms target minimizing pre-defined estimations of the actual cost, called the *cost functions*. Most algorithms are designed only for one or few cost functions, such as the number of gates in the network, and are hard to be adapted for other cost functions. In this section, we propose a way to define customizable cost functions used in resubstitution.

We first define *global cost* and *global cost function* Φ . The global cost is the cost estimation result of the entire network and an indicator of the actual cost. The global cost function Φ is the user-specified function to calculate the global cost . Φ can be arbitrarily defined as long as it satisfies the following two restrictions:

1) Global cost function should be a recursive function. When composing network node by node, the global cost function should be able to calculate the cost after each insertion, given the global cost value of the existing network and a newly-inserted node.
2) Global cost function should only use local variables associated with the newly-inserted node without information from the history of traversal.

In other words, the cost generated at each node contributes directly to the cost evaluation of the entire network. The global cost function specifies how the *contribution* of each node is collected. nd once collected, the temporary result is dissipated and not necessarily memorized. So far, some costs, such as size and MC, can be derived, as each node contributes to global cost individually according to the type of gate that is locally accessible. However, for some estimations, such as depth estimation, local attributes are not sufficient for global cost function Φ .

Therefore, we define *context* and *global context propagation function* (or simply *context function*) Φ . The context of a node n, denoted $_n$, is the variable affecting the global cost function but generated from other parts of the network. It is stored in the memory and assists in the evaluation of the global cost function. The global context propagation function Φ allow users to define how context propagates in the network and save necessary data. We also restrict the context function to a recursive function that can be derived from all the fanins. Note that context can affect the global cost only if it is involved in the global cost function.

Finally, a cost function Φ, as shown in Equation (2), is the combination of two functions.

$$\begin{aligned} \Phi &= (\Phi \ , \Phi \ ) \\ {}' &= \Phi \ (\ , \ _n) \\ _n &= \Phi \ (n, I_n), \end{aligned} \tag{2}$$

where $I_n = \{ \ _i : i \in \delta \ (n)\}$ is the set of context values propagated from node n's fanins, and $'$ are the accumulated global cost before and after consider node n.

fter the cost function is given, we can evaluate the global cost of the whole network using lgorithm 1. In this algorithm, nodes are traversed in topological order so that when processing the context propagation, the required context values I_n are already computed and stored. lso, every node in the network is visited only once to avoid multiple contributions to the global cost. s the number of fanins of a logic gate is usually bounded by a constant, if the cost functions are simple, i.e., are $\mathcal{O}(1)$, then the global cost of a logic network N can be calculated in $\mathcal{O}(|N|)$ operations.

In Table I, we present examples of valid cost definitions to demonstrate the flexibility. The definitions of area (Φ_1) and MC (Φ_2) are straightforward, as no context is required. For depth evaluation (Φ_3), the depth information is recursively derived using context propagation. Sum of depth definition (Φ_4) updates global cost using sum instead of maximum. To calculate the sum of support (Φ_5), the set of support needs to be calculated with the union of fanin supports and then stored to serve its fanouts.

lgorithm 1: Global Cost Calculation

input : Logic network N, global cost function Φ , context function Φ
output: Global cost , context $_n$ for each node n

```
TopologicalSort(G);
  ← 0;
for n ∈ N do
    I_n ← ;
    for i ∈ δ (n) do
        I_n.push( _i);
    end
     _n ← Φ (n, I_n);
      ← Φ ( , _n)
end
return
```

Users can specify more cost functions according to different applications, provided that global cost and context can be derived recursively. It is possible to extend the recursive function to higher-order (propagate context from deeper nodes in TFI instead of only fanins). However, cost functions involving complicated global calculations, such as the sum of all-pairs-min-cut (S PMC) [18], are not compatible with our framework. In Section IV, we will demonstrate its advantages and explain why the restrictions are necessary.

T BLE I: Cost Function Definition Examples

	Cost Name	Cost Functions
Φ_1	rea Cost	$\Phi_1(\ , \ _n) = \ +1$ $\Phi_1(n, I \ ) =$
Φ_2	MC Cost	$\Phi_2(\ , \ _n) = \begin{cases} +1 & n \text{ is ND} \\ & \text{otherwise} \end{cases}$ $\Phi_2(n, I \ ) =$
Φ_3	Depth Cost	$\Phi_3(\ , \ _n) = \max(\ , \ _n)$ $\Phi_3(n, I \ ) = \max_{i \ I_n} \ _i + 1$
Φ_4	Sum of Depth	$\Phi_4(\ , \ _n) = \ + \ _n$ $\Phi_4(n, I \ ) = \max_{i \ I_n} \ _i + 1$
Φ_5	Sum of Support [19]	$\Phi_5(\ , \ _n) = \ +\| \ _n\|$ $\Phi_5(n, I \ ) = \begin{cases} \{n\} & n \text{ PI} \\ \bigcup_{i \ I_n} \ _i & \text{otherwise} \end{cases}$

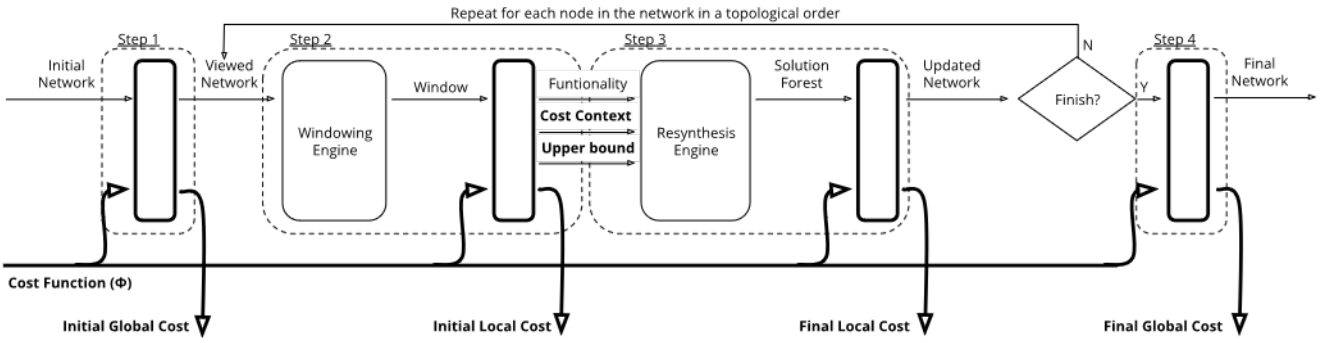


Fig. 1: Cost-generic Resubstitution Flow

IV. Generic Resubstitution Algorithm

Even though the global cost can be evaluated to estimate the actual cost, we cannot optimize it directly. Restricted by the space complexity of truth table operations and the time complexity of finding dependency functions, Boolean method is not affordable to the whole network and algorithms have to optimize global cost by estimating and improving it locally.

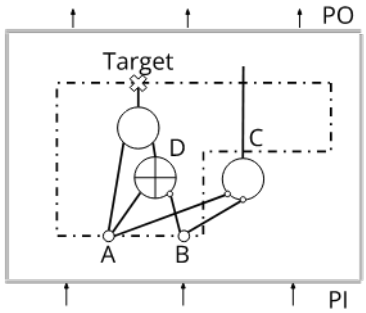
In this section, we describe our cost-generic resubstitution algorithm. The generic resubstitution takes the initial network and a defined cost function as inputs and outputs the final network. s shown in Figure 1, our algorithm can be divided into four steps:

1) The initial network is traversed, the initial global cost is calculated, and the context information is stored.
2) The windowing engine extracts a window from the network, simulates the target function, and constructs a resynthesis problem that contains the context of divisors and an upper bound on the size of the dependency circuit.
3) The resynthesis engine solves the problem and updates the network using the dependency circuit with the lowest cost. The local cost is evaluated before and after resynthesis. Steps 2 and 3 are repeated for each node in the network once in topologically order.
4) The final network is re-evaluated using the cost function, and we get the final global cost.

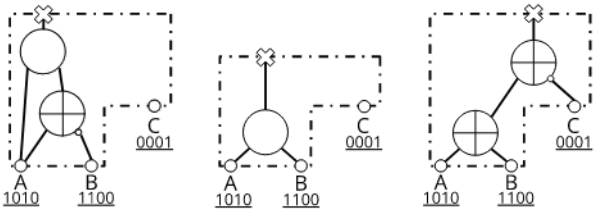
Our algorithm is an extension of the classical resubstitution algorithm described in Section II, which is usually targeted for size (node count) optimization. s marked as bold in Figure 1, the novelties of this work are: (i) defining the *cost-aware resynthesis problem* that brings cost information into the window resynthesis algorithm, and (ii) solving the cost-aware resynthesis problem using the cost-generic searching algorithm. Other steps in the resubstitution algorithm, such as window construction, simulation, and substitution of the resynthesized dependency circuit, are the same as previous works as described in Section II.

. *Local Cost in a Global Context*

In a cost-aware resynthesis problem, both functional equivalence and cost reduction are required. In other words, the resynthesis engine needs to not only check the feasibility of a dependency circuit but also evaluate it in terms of given cost function. Dependency circuit's evaluation is inaccurate if taken out of its context and regarded as a network. For example, in depth optimization, a skewed dependency circuit could be preferred over a balanced one if the level of divisors are different; however, as an entire network, the skewed one has a larger depth than the balanced one.



(a) Window of a target in the logic network



(b) Initial solution (c) Solution 1 (d) Solution 2

Fig. 2: Example of resynthesis problem with multiple solutions

Therefore, we define *local cost in a global context* (or simply *local cost*) ˆ . lgorithm 2 illustrates the procedure of local cost estimation. Similar to lgorithm 1, the cost is evaluated by collecting contributions at each node in topological order. The difference is that only nodes in the MFFC commit to the local estimation because the other nodes will remain in the network no matter they are utilized or not. Therefore, the recursion stops at the divisors and reads the hidden cost directly from the global evaluation. Moreover, during recursive estimation, the context information of the pivot node is also

derived and updated. Figure 2 shows an example of a cost-aware resynthesis problem. s shown in Figure 2b, to run local optimization of the target node, the resubstitution algorithm first finds divisors A and B as the leaves of a reconvergence-driven cut. Node D is in the MFFC because it only supports the target node, while C is collected as a divisor because it is in the window but not the MFFC. Then, while the classical resubstitution algorithm completely isolates its window from the global network, our algorithm allows divisors to obtain context from the external network and updates the context of the root node.

lgorithm 2: Local Cost in Global Perspective

input : Logic network N, target node $n \in N$, set of divisors $S \subset N$, their contexts $\{ _i : i \in S\}$ from global evaluation, cost function $\Phi = (\Phi \ , \Phi \ )$
output: Local cost $\hat{}$, updated context $_n$

```
Function  ompute ostRec( ̂, v):
    if v ∈ S or v is visited then
        return  _v
    end
    I_v ←
    for i ∈ δ (n) do
        I_v.push( ompute ostRec( , i) )
    end
     _v ← Φ (v, I_v)
    ̂ ← Φ ( ̂, v,  _v)
    return  _v
 ̂ ← 0
  _n ←  ompute ostRec( ̂, n)
return ̂
```

With the utilization of global context, our local cost estimation is accurate because of the assumptions and restrictions in Section III. On one hand, local cost evaluation in the MFFC is not affected by nodes in the TFI because the Φ is irrelevant to the history of traversal. On the other hand, divisors provide sufficient context for the root node. s a cut of the root node is included in the divisors' set, the context propagation from PIs to the root node passes through at least one divisor, where contexts of the TFI are accumulated. Notice that in our workflow, the root node of windows are selected in a topological order so that later windows can obtain the updated context of the network. However, when solving one resynthesis problem, its influence of updates on latter resynthesis problems is unpredictable, and the gap between local cost and global cost remains.

B. Cost-generic Searching lgorithm

Given a cost-aware resynthesis problem, *solutions* are dependency circuits. solution is *feasible* if it implements the correct functionality. *Solution forest* is a set of feasible solutions. nd the *optimal solution* is a feasible solution with the lowest local cost in the forest.

The same functionality can be implemented by multiple dependency circuits, and the number of feasible solutions for a given problem might be large. In the example from Figure 2, the target of the resynthesis problem is (1000) and the divisors are $A : (1010)$, $B : (1100)$, $C : (0001)$. Besides the initial solution shown in Figure 2b, we can also derive the same target function using two different dependency functions: $A \wedge B$ and $!A \oplus B \oplus C$. Function $A \wedge B$ implies the dependency circuit in Figure 2c, where blank circles are ND gates and $\oplus$ are XOR gates. Except for Figure 2d, two more dependency circuits can be generated by applying the commutativity law on $!A \oplus B \oplus C$. To be cost-generic, all four solutions need to be collected and evaluated because different cost functions may imply different optimalities. For example, solution 1 is better when optimizing for size, and solution 2 is better if the cost is MC. lso, in depth optimization, the three different structures of $!A \oplus B \oplus C$ could have different costs, depending on the arrival times of A, B, and C. Note that we use X G as the logic network in this example and in the rest of this paper, but our generic resubstitution algorithm, as well as the cost-generic framework, is not limited to X Gs and can be extended to other networks.

lgorithm 3: Generic Resynthesis lgorithm

input : Window W, cost function Φ, upper bound $^{\max}$
output: solution q

```
// Phase 1:
Q ←
for each structure do
    for each input combination of W.divisors do
        q′ ← BuildNetwork()
        f ← Simulate(q′)
        if f = W.target then
            Q.push(q)
        end
    end
end
// Phase 2:
  ←  max
q ←
for q′ ∈ Q do
     ′ ← EvaluateCost(q′, Φ)
    if  ′ <  then
        q ← q′
          ←  ′
    end
end
return q
```

Our cost-generic searching algorithm is shown in lgorithm 3. The algorithm is divided into two phases. In phase 1, the resynthesis engine enumerates all the possible structures with all possible divisor permutations. For each possible circuit, the engine plugs in the divisors' truth table and runs

a simulation. Circuits with correct functionality are collected to a solution forest. Then, in phase 2, we estimate the cost of each candidate solution using lgorithm 2, and return the solution that reduces cost the most. The local cost of the initial solution is set to be the upper bound $_{\max}$ so that the returned dependency circuit is strictly better than the initial circuit.

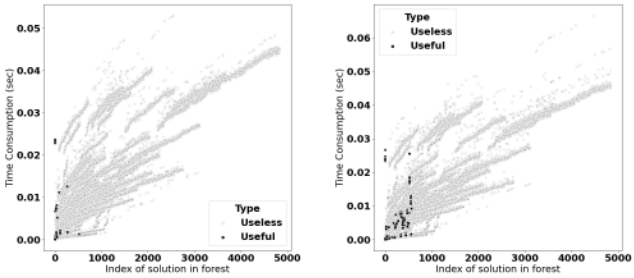
Since phase 2 only returns one solution, whether the optimal solution is contained in the forest determines the *effectiveness* of our algorithm. Meanwhile, the size of the solution space we explored in phase 1 to find the optimal solution determines the *efficiency*. The effectiveness can be improved by extending the search space and enumerating more possible circuits because the solution we collected updates the result only when it has a lower cost. However, this approach would result in poor efficiency.

better search sequence is crucial to improve both effectiveness and efficiency. Table II shows three different searching orders of potential structures with less or equal to three nodes. Structures are sorted in the ascending order of multiplicative complexity, the number of gates, and the complexity of enumerating divisors' permutations, respectively. Note that the size and MC evaluation are context-free, meaning that the cost will monotonically increase in these sequences without knowing the divisors' context information. Since solutions found in the latter structures are worse than the former solutions and will not update the best result, we can terminate the search immediately after seeing the first feasible solution.

monotonic sequence is efficient, because we collect only one solution to the forest and waste no effort on simulation and evaluation of other candidates. However, such monotonicity does not hold for every cost definition, and the majority of practical cost functions, such as depth and support size, depend heavily on the divisors' context.

In this scenario, we define a heuristic sequence based on effort and likelihood of finding the feasible solution, which is expected to be efficient for different costs in general. The sequence, as well as the enumeration complexity of each structure, is shown in Table II. Note that the complexity of a brute-force searching is the number of permutations, and we reduced it using two techniques based on the type of the topmost node: (i) hash table look-up for XORs, and (ii) unateness-based pruning for NDs.

Figure 3 visualizes the distribution of feasible solutions using effort-first-order. Dots in the scatter plot are the feasible solutions, and the coordinates of each node represent the effort it takes to find this solution. The x-axis is the index of this solution in the forest, which indicates the number of other solutions evaluated before it. nd y-axis is the run-time in seconds. The node is colored black if it has the lowest cost, returned in phase 2, and is useful. On the opposite, it is colored grey if it is collected and evaluated but not used. In both plots, we randomly sampled 1M solutions from more than ten thousand resynthesis problems in a dry-run on *voter* circuit [20]. Observe that though the number of feasible solutions can be more than 5000 for some resynthesis problems, the useful solutions are located at the bottom-left



(a) MC optimization (b) Depth optimization

Fig. 3: Distribution of useful solution among all feasible solutions in heuristic effort-first-search

corner of the plot. Therefore, users can set an upper bound on the size of the solution forest, and our resynthesis process can be terminated after the limit is reached without losing too much effectiveness.

V. Experiment l Results

The proposed framework is implemented as part of the C++ logic network library *mockturtle* [21]. Customizable cost functions may be defined using the structure described in Section III. Experiments are run on an MD Ryzen 7 5800H CPU at 3.2GHz with 16 GB of R M. The input size of our window (k) is set to be 8, with the maximum number of divisors (d) equal to 150. We use the benchmarks from EPFL Benchmark Suite [20]. ll results passed the equivalence checking command from BC (*cec*). In Section V- , we demonstrate that the framework is capable of optimizing towards various cost functions. In Section V-B, we compare the effectiveness of our generic resubstitution on the MC cost against the state-of-the-art specialized algorithm optimizing for MC and against a non-specialized algorithm optimizing for size.

. *Cost Functions*

In the first experiment, we optimized 20 circuits using five different cost functions in Table I. In each run, two cost functions are used: the local cost function (Φ_{local}) is passed into the windowing and resynthesis engine for local cost estimation, and the global cost function (Φ_{global}) is used to calculate the cost of the entire circuit before and after the optimization. The script is operated on the circuit only once.

Table III shows the normalized average improvement on all 20 benchmarks. The first five rows in the table represent five separate optimizations, each targeting a cost function from Φ_1 to Φ_5, respectively. The sixth row is a baseline, where no cost function is given, and the resynthesis engine returns the first feasible solution without evaluation. We measure all the five global costs in the column before and after each optimization. Entries in the table are the improvement of geometric mean of cost (i.e. $impr. = 1 \quad \overline{\ }/\overline{\ }$). Positive improvement means the cost is reduced after optimization and vice versa. The result

T BLE II: Enumeration of potential solution using different searching order

(a) Multiplexitive complexity

(b) Gate number

(c) Effort (let d be the number of divisors)

T BLE III: Normalized verage Improvement

	Evaluation Cost Function (Φ_{global})				
Φ_{local}	Φ_1	Φ_2	Φ_3	Φ_4	Φ_5
Φ_1	**19.35%**	29.74%	-5.89%	16.56%	18.14%
Φ_2	13.51%	**49.59%**	-33.58%	-9.31%	13.62%
Φ_3	15.85%	27.03%	**12.17%**	27.40%	16.75%
Φ_4	20.29%	30.76%	-2.73%	**20.86%**	19.48%
Φ_5	19.68%	31.01%	-6.11%	17.07%	**18.50%**
	1.55%	5.63%	-4.13%	1.01%	2.83%

on the diagonal indicates the effectiveness of our algorithm for each cost function.

Results show that our algorithm is cost-generic. For the five selected cost functions, our resubstitution algorithm optimized the circuit accordingly and reduced the costs by 19.35%, 49.59%, 12.17%, 20.86%, and 18.50%, respectively. In each row, the trade-off between different cost objectives can be observed. For example, depth is sacrificed to obtain better size reduction.

Besides, we notice that for Φ_1, Φ_4, Φ_5, the best result is achieved when optimizing each window using different cost functions. It means that the gap between local cost and global cost indeed exists. Even though we take hidden costs into consideration and improve the accuracy of cost estimation, we still cannot fully predict the impact of modifying the network on later optimization. Greedily selecting resynthesis solution is not always the optimal strategy for global optimization.

B. Comparison to Specialized lgorithm

Table IV shows a comparison of our algorithm with two state-of-the-art algorithms on MC optimization problems. The "specialized algorithm" is a cut-rewriting-based algorithm using an MC-oriented database [5]. The "non-specialized algorithm" is the state-of-the-art resubstitution algorithm targeting total gate count [22]. ll three algorithms take the same initial networks and traverse the network only once. The QoR and run-time on all the benchmarks are presented, and the normalized improvement is calculated. Our method achieved the best overall QoR among the three algorithms.

Compared to the non-specialized algorithm for size, our algorithm has a more accurate cost evaluation. Therefore, though both algorithms are able to find the same candidate resynthesis solutions, we make better decisions and choose the solution that reduces cost the most. nd compared to the specialized algorithm, though both algorithms update the local dependency circuit greedily, cut-based methods are limited by the size of the cut and cannot find those solutions based on divisors in a large window.

VI. Conclusions nd future work

In this paper, we consider different target technologies in logic synthesis and propose a generic resubstitution algorithm. Our implementation allows fast prototyping and experimenting with different cost functions. Users only need to describe the cost with a few lines of code, and our algorithm can optimize the circuit accordingly. The cost definition method we propose has good flexibility and is compatible with various optimization objectives. Moreover, with the help of divisors and a more accurate cost estimation, our algorithm can find the appropriate dependency circuit to optimize the total cost. Experiments show that our algorithm can achieve similar or even better results than specialized algorithms in multiplicative complexity optimization.

s future work, we plan to improve the flexibility and efficiency of our algorithm. Currently, cost functions are restricted to be recursively defined, and our algorithm takes much longer to find a resynthesis solution because cost-specific pruning is not applicable. It would be interesting if a more efficient cost-generic searching algorithm could be developed.

T BLE IV: Multiplicative Complexity Optimization

	original		MC-specialized lgorithm (k=4)			Non-specialized lgorithm (k=8)			**ours** (k=8)		
benchmark	ND [#]	Level [#]	ND [#]	time [s]	impr.	ND [#]	time [s]	impr.	ND [#]	time [s]	impr.
dder	1020	255	134	0.05	86.9%	764	0.10	25.1%	255	0.63	75.0%
Barrel Shifter	3336	12	2365	0.25	29.1%	3136	0.53	6.0%	1578	2.87	52.7%
Divisors	57247	4372	36207	4.78	36.8%	33298	5.84	41.8%	17600	23.62	69.3%
Hypotenuse	214335	24801	95557	17.04	55.4%	203842	14.77	4.9%	74984	82.46	65.0%
Log2	32060	444	20190	3.00	37.0%	29154	5.02	9.1%	19260	66.08	39.9%
Max	2865	287	1660	0.30	42.1%	2863	0.25	0.1%	1776	2.36	38.0%
Multiplier	27062	274	14287	2.16	47.2%	24817	4.34	8.3%	13836	48.69	48.9%
Sine	5416	225	3519	0.66	35.0%	4882	1.30	9.9%	3209	19.56	40.7%
Square-root	24618	5058	13307	2.33	45.9%	18348	2.18	25.5%	7041	11.81	71.4%
Square	18484	250	11602	1.65	37.2%	15729	2.39	14.9%	6846	15.54	63.0%
Round-robin rbiter	11839	87	7212	1.08	39.1%	11839	2.19	0.0%	11839	35.77	0.0%
Coding-cavlc	693	16	646	0.26	6.8%	612	0.98	11.7%	456	10.77	34.2%
LU control unit	174	10	121	0.08	30.5%	91	0.10	47.7%	50	1.13	71.3%
Decoder	304	3	304	0.01	0.0%	304	0.08	0.0%	278	6.52	8.6%
i2c controller	1342	20	1086	0.19	19.1%	1207	0.26	10.1%	1082	2.35	19.4%
int to float converter	260	16	217	0.09	16.5%	220	0.50	15.4%	183	2.39	29.6%
Memory controller	46836	114	36398	3.51	22.3%	44448	8.50	5.1%	37092	72.07	20.8%
Priority Encoder	978	250	428	0.09	56.2%	586	0.10	40.1%	643	1.55	34.3%
Lookahead XY router	257	54	196	0.04	23.7%	229	0.04	10.9%	165	0.40	35.8%
Voter	13758	70	5437	1.19	60.5%	7413	1.49	46.1%	3033	7.76	78.0%
geo.mean	4364	136	2595	0.47		3568	0.82		2197	7.70	
normalized impr.	1.000				**40.5%**			**18.2%**			**49.6%**

REFERENCES

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

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

[3] . Mishchenko, S. Chatterjee, and R. K. Brayton, "D G-aware IG rewriting: fresh look at combinational logic synthesis," in *Proceedings of the 43rd Design utomation Conference, D C 2006, San Francisco, C , US , July 24-28, 2006*, 2006, pp. 532–535.

[4] S. Chatterjee, . Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *ICC D-2005. IEEE/ CM International Conference on Computer- ided Design, 2005.*, 2005, pp. 519–526.

[5] E. Testa, M. Soeken, L. G. marù, and G. De Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *Proceedings of the 56th nnual Design utomation Conference 2019, D C 2019, Las Vegas, NV, US , June 02-06, 2019*. CM, 2019, p. 74.

[6] G. Meuli, M. Soeken, and G. De Micheli, "Xor- nd-Inverter graphs for quantum compilation," *npj Quantum Information*, vol. 8, no. 1, pp. 1–11, 2022.

[7] C. Schnorr, "The multiplicative complexity of boolean functions," in *pplied lgebra, lgebraic lgorithms and Error-Correcting Codes, 6th International Conference, ECC-6, Rome, Italy, July 4-8, 1988, Proceedings*, ser. Lecture Notes in Computer Science, T. Mora, Ed., vol. 357. Springer, 1988, pp. 45–58.

[8] E. Testa, M. Soeken, H. Riener, L. G. marù, and G. De Micheli, " logic synthesis toolbox for reducing the multiplicative complexity in logic networks," in *2020 Design, utomation & Test in Europe Conference & Exhibition, D TE 2020, Grenoble, France, March 9-13, 2020*. IEEE, 2020, pp. 568–573.

[9] S. Lee, H. Riener, and G. De Micheli, "Logic resynthesis of majority-based circuits by top-down decomposition," in *24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2021, Vienna, ustria, pril 7-9, 2021*, M. Shafique, . Steininger, L. Sekanina, M. Krstic, G. Stojanovic, and V. Mrazek, Eds. IEEE, 2021, pp. 105–110.

[10] I. Hálecek, P. Fiser, and J. Schmidt, " re XORs in logic synthesis really necessary?" in *20th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2017, Dresden, Germany, pril 19-21, 2017*, 2017, pp. 134–139.

[11] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPG mapping solution," in *Proceedings of the 1999 CM/SIGD Seventh International Symposium on Field Programmable Gate rrays, FPG 1999, Monterey, C , US , February 21-23, 1999*, 1999, pp. 29–35.

[12] . Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," 2006.

[13] C. Yu, M. J. Ciesielski, and . Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *IEEE Trans. Comput. ided Des. Integr. Circuits Syst.*, vol. 37, no. 9, pp. 1907–1911, 2018.

[14] L. maru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: new paradigm for logic optimization," *IEEE Transactions on Computer- ided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2015.

[15] H. Riener, W. Haaswijk, . Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and D G-aware: Rewriting boolean networks with exact synthesis," in *Design, utomation & Test in Europe Conference & Exhibition, D TE 2019, Florence, Italy, March 25-29, 2019*, 2019, pp. 1649–1654.

[16] H. Riener, . Mishchenko, and M. Soeken, "Exact D G-aware rewriting," in *2020 Design, utomation & Test in Europe Conference & Exhibition, D TE 2020, Grenoble, France, March 9-13, 2020*, 2020, pp. 732–737.

[17] P. Bjesse and . Borälv, "D G-aware circuit compression for formal verification," in *2004 International Conference on Computer- ided Design, ICC D 2004, San Jose, C , US , November 7-11, 2004*, 2004, pp. 42–49.

[18] P. Kudva, . Sullivan, and W. Dougherty, "Metrics for structural logic synthesis," in *IEEE/ CM International Conference on Computer ided Design, 2002. ICC D 2002.*, 2002, pp. 551–556.

[19] L. Machado and J. Cortadella, "Support-reducing decomposition for fpga mapping," *IEEE Transactions on Computer- ided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 213–224, 2020.

[20] L. G. marù, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," 2015.

[21] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL Logic Synthesis Libraries," *arXiv:1805.05121 [cs]*, Nov. 2019, arXiv: 1805.05121. [Online]. vailable: http://arxiv.org/abs/1805.05121

[22] H. Riener, S.-Y. Lee, . Mishchenko, and G. De Micheli, "Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis," in *2022 27th sia and South Pacific Design utomation Conference (SP-D C)*, 2022, pp. 395–402.