DCLOG: Don't Cares-based Logic Optimization using Pre-training Graph Neural Networks

Rongliang Fu[1], Libo Shen[1], Ziyi Wang[1], Zhengxing Lei[2], Zixiao Wang[1], Junying Huang[3*], Bei Yu[1] and Tsung-Yi Ho[1]
[1]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*, Hong Kong, China
[2]*Faculty of Electronic and Information Engineering, Xi'an Jiaotong University*, Xi'an, China
[3]*SKLP, Institute of Computing Technology, CAS*, Beijing, China

***Abstract*—Logic rewriting serves as a robust optimization technique that enhances Boolean networks by substituting small segments with more effective implementations. The incorporation of don't cares in this process often yields superior optimization results. Nevertheless, the calculation of don't cares within a Boolean network can be resource-intensive. Therefore, it is crucial to develop effective strategies that mitigate the computational costs associated with don't cares while simultaneously facilitating the exploration of improved optimization outcomes. To address these challenges, this paper proposes DCLOG, a don't cares-based logic optimization framework, to efficiently and effectively optimize a given Boolean network. DCLOG leverages a pre-trained graph neural network model to filter out cuts without don't cares and then performs an incremental window simulation to calculate don't cares for each cut. Experimental results demonstrate the effectiveness and efficiency of DCLOG on large Boolean networks, specifically average size reductions of 15.64% and 1.44% while requiring less than 23.84% and 44.70% of the average runtime compared with state-of-the-art methods for the majority-inverter graph (MIG), respectively.**

***Index Terms*—logic optimization, don't cares, graph neural network, majority-inverter graph**

I. Introduction

Logic optimization [1], [2] is a crucial step in realizing efficient digital systems. Among various logic optimization techniques, logic rewriting stands out for its efficiency and effectiveness in optimizing a Boolean network. Given an input Boolean network, logic rewriting can streamline the logic structure, resulting in less depth and size while maintaining the original function. Various methods exist for logic rewriting. Notable approaches include DAG-aware rewriting [3], mapping-based rewriting [4], and cut-based rewriting [5]. Among them, a key component is SAT-based exact synthesis [6], which finds the most optimized substitution structure for a given Boolean network. Typically, optimal structures are pre-computed and saved in a database due to the computational complexity of exact synthesis. Hence, SAT-based exact synthesis methods are extremely popular due to their well-optimized results.

On the other hand, don't cares play a crucial role in optimizing logic networks, which can simplify Boolean expressions by allowing designers to freely choose output values for specific input patterns [7]. Generally, don't cares are incorporated with exact synthesis and Boolean matching, providing flexibility in selecting replacement candidates with higher optimization potential. The state-of-the-art logic optimization methods have utilized the advantageous features of don't cares, such as logic resubstitution [8] and logic rewriting [5]. These methods construct cuts for each gate through cut enumeration and then iteratively perform the Boolean function retrieval in the database by Boolean matching [9] for a better implementation for each cut. Due to the application of don't cares, the Boolean function retrieval can find more implementations that contain given simplified Boolean functions from a database, which provides a potential for exploring its optimal implementation. These state-of-the-art methods can significantly optimize a given Boolean network compared with logic rewriting without don't cares, especially for the majority-inverter graphs (MIGs) [10].

Although logic rewriting methods with don't cares have achieved significant success in optimizing the size and depth of Boolean networks, they require substantial computational costs. Taking controllability don't cares (CDCs) as an example, its calculation requires the determination of the transitive fanin of nodes in a given cut and then a functional simulation from these transitive fanin to all nodes in the cut. In fact, this process is very time-consuming. Furthermore, Boolean networks typically contain a large number of nodes, each with multiple cuts, which further increases the computational costs of don't cares. In our cases, the calculation [10] of don't cares nearly increased 6× runtime of the logic rewriting process on large Boolean networks, underscoring the need for strategies to reduce computation time in don't care-based optimization.

To address this issue, we propose DCLOG, an efficient don't care-based logic optimization framework. While maintaining the superiority of the resulting Boolean network in size and depth, DCLOG employs two main strategies to minimize the computation cost. First, the calculation occurrence of don't care will be reduced by filtering potential cuts without don't cares. DCLOG uses pre-training graph neural networks to predict the output probability of each node within a given Boolean network, subsequently utilizing these probabilities to filter cuts. Then, the calculation cost of don't care will be reduced by dynamic simulation. Due to high computational demands, the calculation of don't cares is usually confined to a local Boolean network, where a Boolean simulation is required for exact don't care results. DCLOG uses the leaves of cuts as boundaries to facilitate incremental simulation of the local network, thereby significantly reducing the overall simulation costs. We evaluate our DCLOG on large Boolean networks in the EPFL and IWLS 2005 benchmarks to demonstrate its effectiveness and efficiency.

Overall, the contributions of this paper are as follows:

- We propose an efficient don't care-based logic rewriting framework to achieve superior Boolean logic optimization.
- We develop an effective cut-filtering method utilizing a pre-trained graph neural network model, efficiently excluding cuts without don't cares.
- We introduce an incremental simulation method designed to significantly reduce unnecessary simulation overhead.
- Experimental results on large-scale EPFL and IWLS 2005 benchmarks show that DCLOG outperforms state-of-the-art logic rewriting and resubstitution methods in runtime and size reduction for MIG optimization.

II. Background

A. Don't Cares

In digital logic, don't cares of a Boolean function refer to input patterns for which the function's output is irrelevant or that never occur. This flexibility allows certain inputs to be assigned any

* Corresponding author: huangjunying@ict.ac.cn.

value without impacting the circuit's primary outputs, enabling the transformation of a Boolean function f into an equivalent function f' without altering its intended behavior. Don't cares facilitate more efficient Boolean matching by offering added flexibility.

CDCs, as a type of don't cares, represent input patterns that never occur in a Boolean logic network. Considering that CDCs possess relatively acceptable computational complexity and effectiveness for logic optimization among various types of don't cares, this paper selects CDCs for logic rewriting with don't cares. Given the Boolean function f with its inputs $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and a set of additional variables $\mathbf{v}$, where $\mathbf{v} \cap \mathbf{x} = \emptyset$ and $\mathbf{v}$ sources from transitive fanin of nodes in $\mathbf{x}$, the CDCs of f are defined as follows:

$$\mathrm{CDC}(f) = \{\mathbf{x} \mid C_f(\mathbf{x}) = 1\}, \tag{1}$$

where the controllability condition $C_f(\mathbf{x})$ is given by

$$C_f(\mathbf{x}) = \forall \mathbf{v}, \bigvee_{x_i \in \mathbf{x}} x_i \oplus f_{x_i}(\mathbf{v}, \mathbf{x}), \tag{2}$$

and $f_{x_i}(\mathbf{v}, \mathbf{x})$ is the Boolean function describing the behavior of x_i in terms of the variables $\mathbf{v} \cup \mathbf{x}$.

We take the Boolean function $f = (d \wedge b) \vee (b \wedge c) \vee (d \wedge c)$ as a running example. The Boolean function f's inputs are $\mathbf{x} = \{b, c, d\}$, where d's corresponding Boolean function $f_d = a \wedge b$, and $\mathbf{v} = \{a\}$. According to Equation (2), $C_f(\mathbf{x}) = \forall a, (b \oplus b) \vee (c \oplus c) \vee (d \oplus (a \wedge b)) = \forall a, d \oplus (a \wedge b) = d \wedge (\neg b)$. Hence, the set of CDCs for f is $\{(b = 0, c = 0, d = 1), (b = 0, c = 1, d = 1)\}$. Utilizing these don't cares, the Boolean function $f = (d \wedge b) \vee (b \wedge c) \vee (d \wedge c)$ can be simplified in a more compact form $f = d \vee (b \wedge c)$.

B. Logic Rewriting with Don't Cares

Several established logic rewriting frameworks have achieved remarkable optimization results. Calvino *et al.* proposed map-based logic optimization techniques [4], while Mishchenko *et al.* introduced DAG-aware logic rewriting [3]. Additionally, Lee *et al.* developed a heuristic logic resubstitution method [8]. More recently, Calvino *et al.* presented a Boolean matching approach utilizing don't cares [5], which achieves a significant reduction in circuit size compared with other logic rewriting strategies.

In general, logic rewriting with don't cares involves three main steps: (i) partitioning a Boolean network into subsets by constructing multiple cut sets for each gate, (ii) calculating don't cares for each cut, and (iii) performing Boolean matching with these don't cares. However, this approach faces two significant challenges: ❶ The calculation of don't cares for each cut is computationally expensive; ❷ Many cuts in Boolean matching do not actually contain don't cares, making these calculations often redundant.

To address these challenges, our proposed method introduces a novel strategy to selectively identify cuts that have a high probability of possessing don't cares. Furthermore, since only partial simulation results are typically required for don't care calculation, we optimize the simulation process to improve efficiency.

C. Circuit Representation Learning

Circuit representation has been widely explored in the deep learning community, with various advanced graph representation methods, such as those in [11]–[14], providing valuable node embeddings for our research. The quality of these embeddings heavily depends on the capacity of the embedding model, which in turn determines prediction accuracy, particularly in identifying don't-care conditions. Significant efforts have been made to improve model capacity. For instance, DeepGate [13] uses logic-1 probability supervision for circuit representation, while its successor, DeepGate2 [14], distinguishes between structural and functional similarities during learning. However, their restrictive loss functions struggle to efficiently capture the logic network's structure, limiting prediction accuracy. In contrast, FGNN [11] leverages contrastive learning to model circuit functionalities. Its enhanced version, FGNN2 [12], further improves performance by introducing a specialized contrastive loss to quantify functional differences and an order-invariant encoding scheme. This enables highly accurate simulation results within seconds via inference from a pre-trained model. Given these advantages, we adopt FGNN2 for node embedding extraction in this work.

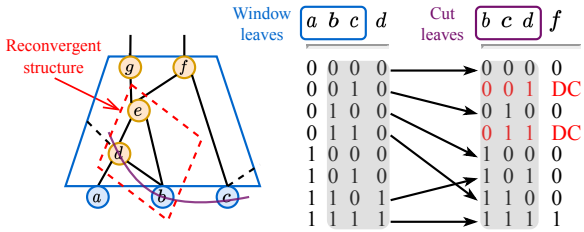

Fig. 1. The calculation of don't cares via window simulation, where a, b, and c are window inputs, and d, e, f, and g are AND gates.

III. Problem Formulation

A. Terminology

A Boolean function is a mapping from a n-dimensional Boolean space into a 1-dimensional one: $\{0,1\}^n \rightarrow \{0,1\}$. A Boolean function can be instantiated as a Boolean network, a directed acyclic graph $N(V, E)$, where each node $v \in V$ corresponds to a logic gate, and each directed edge $(u, v) \in E$ is a wire connecting node u with node v. Moreover, for a node $v \in V$, TFI(v) is the set of its transitive fan-in. For any node $u \in$ TFI(v), there is a path from u to v.

The simulation result for a gate in the Boolean network is a sequence of $0s$ and $1s$, representing the output under all possible input patterns. For a node v, we defined pattern s_v as the sequence of $0s$ and $1s$ under the simulation. $|s_v|$ represents the frequency of 1's in the simulation result sequence. In Fig. 1, the simulation pattern for node d is $s_d = 00000011$, and $|s_d| = 0.25$.

Definition 1 (Cut)**.** *A cut $\mathcal{C}$ in a Boolean network N is defined as a pair $(r, \mathcal{C}_l)$, where r is its root node, $\mathcal{C}_l$ represents the set of its leaves, and any path from a primary input (PI) of N to r must pass through at least one node in $\mathcal{C}_l$.*

The root node r serves as the output of the cut, and each leaf $l \in \mathcal{C}_l$ serves as an input of the cut. So, the Boolean function corresponding to the cut $\mathcal{C}$ can be represented as $f(\mathcal{C}_l)$. $K = |\mathcal{C}_l|$ is the cut size. Notably, for a node $v \in V$, it may have multiple cuts, and the set of its cuts is denoted as $\mathbb{C}$. In Fig. 1, node f has a cut $\mathcal{C}$ with $\mathcal{C}_l = \{b, c, d\}$. Similarly, a window $\mathcal{W}$ is defined analogously to a cut but may contain multiple outputs. $\mathcal{W}_l$ denotes the leaves of a window, and $L = |\mathcal{W}_l|$ is the window size. Besides, $\mathcal{W}$ also represents the set of all nodes within a window, containing root nodes and their multi-level bounded transitive fanin/fanout nodes. In Fig. 1, the blue trapezoid represents a window $\mathcal{W}$ with leaves $\mathcal{W}_l = \{a, b, c\}$.

Definition 2 (Containment)**.** *The containment relationship exists between a cut $\mathcal{C}$ and a window $\mathcal{W}$ if $\mathcal{C}$ and $\mathcal{W}$ satisfy $\mathcal{C}_l \not\subset \mathcal{W}_l \wedge \mathcal{C}_l \subset \mathcal{W}$. That is, cut $\mathcal{C}$ is contained in window $\mathcal{W}$.*

Definition 3 (Reconvergence)**.** *In a Boolean network, a path is a finite sequence of connected nodes, with the start node v_s and the end node v_t. Two paths are considered reconvergent if they start at the same node v_s and end at the same node v_t, respectively.*

In Fig. 1, the paths $b \rightarrow d \rightarrow e$ and $b \rightarrow e$ are reconvergent,

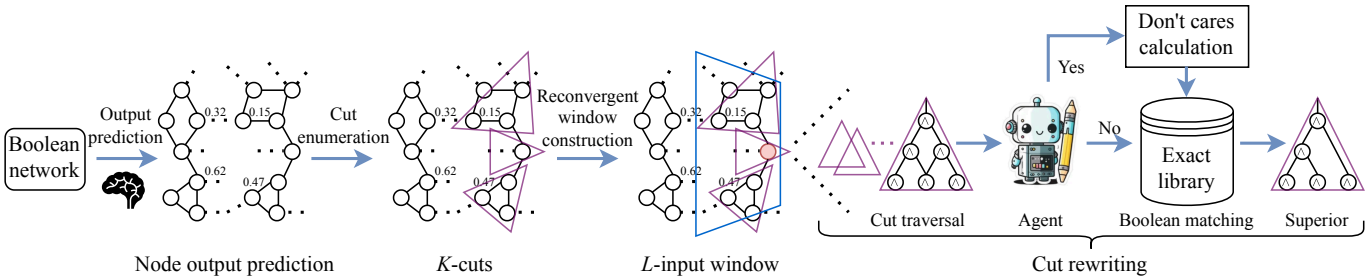

Fig. 2. The workflow of DCLOG.

where the diamond-shaped reconvergent structure wrapped in a red dotted diamond is essential for the reconvergent path. If a window $\mathcal{W}$ contains reconvergent paths, we call it a reconvergent window [15]. Reconvergent paths are crucial for identifying don't cares [15]. Tree-like (non-reconvergent) structures do not contain don't cares in the local space of the node, meaning don't cares may not provide further optimization benefits for such structures [16].

Definition 4 ($\mathcal{NPN}$-equivalence)**.** *Two Boolean functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are considered $\mathcal{NPN}$-equivalent if g can be derived from f through a combination of the following transformations: input inversion N_i (replacing x_i with $\neg x_i$), input permutation P_i (swapping x_i and x_j), and output inversion N_o (replacing f with $\neg f$). Under these transformations, f and g are Boolean equivalent [9].*

For n inputs, there are 2^{2^n} possible Boolean functions. These functions can be grouped into significantly fewer $\mathcal{NPN}$ classes. Specifically, the number of $\mathcal{NPN}$ classes for n-input Boolean functions is $\{14, 222, 616126\}$ for $n = 3$, 4, and 5, respectively.

B. Problem Formulation

This paper focuses on how to optimize Boolean networks more efficiently through don't cares. Given an input Boolean network, the objective is to minimize its size and depth while maintaining functionality. The problem is formulated as:

- **Input**: A given Boolean network $N(V, E)$.
- **Output**: An optimized Boolean network $N'(V', E')$.
- **Constraints**:
  1) Cut size limitation: $\forall \mathcal{C}, |\mathcal{C}_l| \leq K$.
  2) Window size limitation: $\forall \mathcal{W}, |\mathcal{W}_l| \leq L$.
  3) The logic function stays unchanged after optimization.

  Constraints 1 and 2 ensure both the efficiency and feasibility of logic rewriting.
- **Goal**: Minimize the network's size $|V'|$ and depth.

IV. DCLOG

Currently, the state-of-the-art logic rewriting techniques primarily combine don't cares with Boolean matching to discover more potential alternative structures. During this process, the calculation of don't cares for a cut $\mathcal{C}$ usually employs the window simulation approach and mainly consists of two steps: (i) constructing a reconvergent window $\mathcal{W}$ such that $\mathcal{W}$ contains $\mathcal{C}$, followed by simulating this window from its leaves to roots, and (ii) analyzing the simulation results to identify missing input patterns in $\mathcal{C}_l$, which constitute the don't cares for cut $\mathcal{C}$. However, we have identified two key observations: ❶ only a small portion of the simulation results is necessary for calculating don't cares, rendering many of window simulation results superfluous and time-consuming; and ❷ most cuts do not contain don't cares, making the don't care calculation for these cuts inefficient and unnecessary.

In light of these discoveries, this section proposes a highly efficient logic rewriting framework with don't cares, whose flow is shown in Fig. 2. Initially, preparatory tasks include the construction of an exact database for Boolean matching and the training of a node output prediction model. The exact database is enhanced with additional don't cares information and is organized by $\mathcal{NPN}$ classes, each associated with don't cares [5], as detailed in Section IV-D. The node output prediction model uses the node embedding generated by the FGNN2 model as input to predict the probability of a node outputting logic '1', as detailed in Section IV-C. Subsequently, the process of logic rewriting for a given Boolean network proceeds as Fig. 2.

A. Don't Care Calculation and Incremental Simulation

We first detail how DCLOG calculates don't cares of a cut $\mathcal{C}$ with the Boolean function $f(\mathcal{C}_l)$. First, according to Equation (1), the calculation of CDCs of cut $\mathcal{C}$ requires additional transitive fanin $\mathbf{v}$. Due to the universal quantification about $\mathbf{v}$ required in Equation (2), all output patterns of each node in $\mathcal{C}_l$ are required. Due to high computational costs, it's impractical to obtain the global output patterns of each node in terms of the primary inputs of the Boolean network. So, a L-input reconvergent window $\mathcal{W}$ is constructed for the root r of cut $\mathcal{C}$ to limit the transitive fanin of node r. When window $\mathcal{W}$ contains cut $\mathcal{C}$, the leaves $\mathcal{W}_l$ of window $\mathcal{W}$ can become the additional transitive fanin $\mathbf{v}$ of the Boolean function $f(\mathcal{C}_l)$, which indicates the computability of don't cares. So, the simulation for window $\mathcal{W}$ is performed from its leaves $\mathcal{W}_l$ to cut leaves $\mathcal{C}_l$, thereby generating output patterns of the cut leaves $\mathcal{C}_l$. According to these output patterns, CDCs are collected by identifying which input patterns do not appear at $\mathcal{C}_l$. This process, known as the projection of don't cares, has a complexity that grows exponentially with $|\mathcal{W}_l|$.

Fig. 1 illustrates an example of the don't care calculation, where nodes d, e, f, and g represent the $\wedge$ operator. The node f has a cut $\mathcal{C}$, with its leaves $\mathcal{C}_l = \{b, c, d\}$ marked by a purple arc. First, a reconvergent window $\mathcal{W}$ with leaves $\mathcal{W}_l = \{a, b, c\}$, represented by the blue trapezoid, is built from the root f, where d is contained within $\mathcal{W}$ and $\mathbf{v} = \{a\}$. Next, the simulation is performed for window $\mathcal{W}$ using the input patterns of a, b, and c, yielding output patterns s_b, s_c, and s_d. The tables on the right display these simulation results. Then, we examine the simulation results for each cut leaf $l \in \mathcal{C}_l$ to identify any missing input patterns, which constitute the don't cares for cut $\mathcal{C}$. In this example, cut leaves $\mathcal{C}_l$ lack the input patterns $\{(b = 0, c = 0, d = 1), (b = 0, c = 1, d = 1)\}$, which are thus regarded as the don't cares, i.e., $dc_f = \{(b = 0, c = 0, d = 1), (b = 0, c = 1, d = 1)\}$.

During the above process, the window simulation makes up one of the heaviest workloads for the calculation of don't cares. The key insight is that we can bypass simulating the entire window and instead focus solely on the simulation results of the leaves of cut $\mathcal{C}$, since the calculation of don't cares only requires examining the input patterns of cut $\mathcal{C}$. In our Boolean network structure, each node is

assigned a serial number, and all nodes are stored in topological order within an array. The window simulation also follows this topological order, meaning that nodes with higher serial numbers depend on the simulation results of nodes with lower serial numbers, while the latter are independent of those with higher serial numbers. Hence, we introduce the incremental simulation to reduce computational costs. The serial number of the node simulated last is stored, and then the simulation is performed only if the new node's serial number is greater than the last one. By leveraging this incremental approach, we only focus on the output patterns of the $\mathcal{C}_l$ within $\mathcal{W}$. This method reduces the average time by 24.03%, compared with the original approach, which simulates the entire window.

B. Cut Filtering

Although we have achieved efficient don't care calculations through incremental simulation, the computational complexity of this process remains high, particularly for large Boolean networks with many nodes and multiple cuts associated with a single node. Notably, cuts without don't cares, as generated through cut enumeration and contained in the window of its root, can account for up to 73.74%, making it essential to filter these cuts before proceeding with don't care calculations. In this section, we introduce a cut-filtering method designed to minimize unnecessary don't care calculations for cuts.

The cut filtering of DCLOG contains the following two strategies:

1. Filtering Cuts Using Reconvergent Window. Our first strategy aims to identify whether the computability conditions of don't cares are satisfied. According to Equation (2), the calculation of CDCs of cut $\mathcal{C}$ requires additional transitive fanin $\mathbf{v}$. Due to the limitation of computational resources, it's impractical to obtain the global output patterns of each node in terms of the primary inputs of the Boolean network. Hence, Section IV-A creates a reconvergent window for each node to calculate its cuts' don't cares. That is because a reconvergent window can provide the additional transitive fanin while enabling a more effective capture of don't cares than a tree-structured window.

Therefore, the computability conditions of don't cares for a cut $\mathcal{C}$ can be specified as whether the containment relationship exists between cut $\mathcal{C}$ and reconvergent window $\mathcal{W}$. Specifically, we enable the don't care calculation of cut $\mathcal{C}$ when cut $\mathcal{C}$ is contained in window $\mathcal{W}$, *i.e.*, satisfying:

$$\mathcal{C}_l \not\subset \mathcal{W}_l \wedge \mathcal{C}_l \subset \mathcal{W}. \tag{3}$$

Referring to the example in Fig. 1, the cut $\mathcal{C}$ with leaves $\mathcal{C}_l = \{b, c, d\}$ and the window $\mathcal{W}$ with leaves $\mathcal{W}_l = \{a, b, c\}$ are built from the same root f, respectively. Since $\{b, c, d\} \not\subset \{a, b, c\}$ and $\{b, c, d\} \subset \mathcal{W}$, the cut $\mathcal{C}$ is contained in $\mathcal{W}$, that is, cut $\mathcal{C}$ satisfies the computational conditions of CDCs.

2. Filtering Cuts Using Statistical Analysis. Our second strategy aims to determine whether the calculation of don't cares is necessary. As the introduction of the reconvergent window for simulation, we enhance the chances of identifying cuts with don't cares. However, the proportion of cuts that lack don't cares remains significantly high, even reaching up to 73.74% on certain Boolean networks. To address this, we propose the other filtering strategy based on the following observation. In the example in Fig. 1, we observe that the output pattern s_d of leaf d exhibits a low value of $|s_d| = 2/8 = 0.25$, indicating that the full range of input patterns to the cut is not achievable. In fact, a relatively high or low value of $|s_l|$ for any leaf $l \in \mathcal{C}_l$ strongly suggests the presence of don't cares in $\mathcal{C}$, as it implies that $\mathcal{C}_l$ may lack some input patterns. This insight forms the foundation of our second cut-filtering strategy.

Specifically, assuming that output patterns of all cut leaves have been obtained, $\mathcal{C}$ is identified to possess don't cares if its leaves $\mathcal{C}_l$ satisfy the following condition:

$$\exists l \in \mathcal{C}_l, |s_l| > \alpha \vee |s_l| < 1 - \alpha = \textbf{true}, \tag{4}$$

where α is a probability threshold. As α approaches 0 or 1, the cut filtering becomes stricter. The application of the second strategy can effectively filter out many cuts without don't cares (up to 98.72%). However, as discussed in Section IV-A, obtaining output patterns of all cut leaves through simulation requires significant computational costs. Hence, a node output probability prediction model with fast inference is required to make this approach efficient.

C. Node Output Prediction

Our second cut-filtering strategy requires the output probability (i.e., the probability of being logic '1') of each node. However, obtaining these results using traditional circuit simulation methods is prohibitive due to the huge computational complexity. To address this, a model capable of efficiently inferring the output probability of each node in a given Boolean network is essential.

The output probability of a node is inherently linked to its logical function, underscoring the need for a functional representation of each node as input to our predictive model. To address this, we select FGNN2 [12] as our circuit encoder, as detailed in Section II-C. This choice is motivated by its high efficiency and robust capability to capture the logical functionality of circuits through node embeddings.

In FGNN2, the initial step involves converting an input Boolean network into a heterogeneous graph featuring a single node type and two distinct edge types. Taking the AND-inverter graph (AIG) as an example, each AND gate is depicted as a node, while inverters are depicted on the edges, distinguished by the edge type as either an inverter or a non-inverter. FGNN2 then implements an asynchronous message-passing mechanism designed to mirror the logical computation dynamics inherent in gate-level circuits. This message-passing approach in FGNN2 initiates by propagating node features (*i.e.*, input patterns) from primary inputs of the netlist. The propagation follows a topological order, progressing level by level until it encompasses all nodes. During this propagation, FGNN2 uses two distinct multilayer perceptrons (MLPs) as aggregators to independently learn the logical functions of the AND and inverter gates.

Formally, the message aggregating scheme of a node v can be stated as follows:

$$\begin{aligned} \boldsymbol{m}_v^i &= \mathcal{A}^{inv}(\{\boldsymbol{h}_u : u \in \mathcal{P}^i(v)\}) \\ &= \sigma(\mathrm{MLP}^{inv}(\{\boldsymbol{h}_u | u \in \mathcal{P}^i(v)\})); \\ \boldsymbol{h}_v &= \mathcal{A}^{and}(\boldsymbol{m}_v^i, \{\boldsymbol{h}_u : u \in \mathcal{P}^n(v)\}) \\ &= \sigma(\mathrm{MLP}^{and}(f(\boldsymbol{m}_v^i, \{\boldsymbol{h}_u : u \in \mathcal{P}^n(v)\}))), \end{aligned} \tag{5}$$

where $\mathcal{P}^i(v)$ refers to the set of predecessor nodes connected to node v via an inverter edge, while $\mathcal{P}^n(v)$ refers to the set of predecessor nodes connected to node v via a non-inverter edge. The terms $\mathcal{A}^{and}$ and $\mathcal{A}^{inv}$ represent the learnable aggregators specifically designed for AND gates and inverters, respectively. Furthermore, MLP^{and} and MLP^{inv} denote the multilayer perceptrons used in the model. The activation function used here is σ, which is a leaky-ReLU function [17]. The function f serves as the combination function and is implemented as the mean operator, which helps in averaging the features or signals processed through the network.

The above method allows the functionality of each node to be seamlessly integrated into its embedding, enhancing the model's ability to interpret and analyze the circuit's behavior effectively. An MLP model is then applied to predict the output probability $P_v = \mathrm{MLP}(\boldsymbol{h}_v)$ of a node $v \in V$ based on its embedding $\boldsymbol{h}_v$.

Algorithm 1: DCLOG algorithm.

Input: Boolean network $N(V, E)$, cut size K, window size L.
Output: Optimized Boolean network.

1 $P \leftarrow$ predict the output probability of each node in network N.
2 **for** *node* $v \in V$ *in topological order* **do**
3 $original \leftarrow$ calculate the level of node v.
4 $critical \leftarrow$ determine if node v is on a critical path.
5 $\mathcal{W} \leftarrow$ construct a L-input reconvergent window for node v.
6 $S \leftarrow \emptyset$, $G \leftarrow \emptyset$, $best \leftarrow \{original, 0\}$.
7 **for** $\mathcal{C} \leftarrow$ *enumerate* K*-cuts of node* v **do**
8 $dc \leftarrow \emptyset$.
9 **if** $\mathcal{C}_l \not\subset \mathcal{W}_l \wedge \mathcal{C}_l \subset \mathcal{W}$ **then**
10 **if** $\vee_{l \in \mathcal{C}_l}(P[l] > \alpha \vee P[l] < (1-\alpha))$ **then**
11 $S \leftarrow$ incrementally simulate the window $\mathcal{W}$ from $\mathcal{W}_l$ to $\mathcal{C}_l$ using S.
12 $dc \leftarrow$ calculate don't cares for cut $\mathcal{C}$ on window $\mathcal{W}$ using S.
13 $Gs \leftarrow$ perform Boolean matching for cut $\mathcal{C}$ with dc and its truth table in the exact database.
14 **for** $G' \in Gs$ **do**
15 $cur \leftarrow$ calculate the gain from replacing $\mathcal{C}$ with G'.
16 **if** ❶$(critical \wedge cur.l \leq best.l \wedge cur.n > best.n) \vee$
17 ❷$(\neg critical \wedge cur.n > best.n) \vee$
18 ❸$(cur.n = best.n \wedge cur.l < best.l)$ **then**
19 $G \leftarrow G'$, $best \leftarrow cur$.
20 **if** $best.n > 0 \vee (best.n = 0 \wedge best.l < original)$ **then**
21 $N \leftarrow$ replace $\mathcal{C}$ with G in network N.
22 $P \leftarrow$ update the output probabilities of added nodes.
23 **return** N.

D. Boolean Matching with Exact Library and Don't Cares

After identifying a cut $\mathcal{C}$ with don't cares, it is crucial to efficiently retrieve an optimal replacement from the exact database constructed via exact synthesis. This database is organized into $\mathcal{NPN}$-equivalence classes using the method described in [5]. To enable Boolean matching with don't cares, we extend the exact database by incorporating don't cares. For each $\mathcal{NPN}$ class, its don't cares are determined by analyzing functions with lower area costs, applying transformations, and evaluating dominance.

After obtaining the exact database, Boolean matching with don't cares can be performed for cut $\mathcal{C}$. The process begins by converting $\mathcal{C}$'s Boolean function $f(\mathcal{C})$ into a canonical form. This involves locating the lexicographically smallest truth table within its $\mathcal{NPN}$ class, resulting in a class representative f_c along with its associated permutation and negation vectors. The don't cares are then adjusted according to the new permutation, while input and output negations are ignored since they do not affect the don't care conditions.

Using the adjusted don't cares of f_c, potential alternative structures Gs are accessed from the exact database. Each structure $G' \in Gs$ includes its don't cares t, an $\mathcal{NPN}$ class representative, and the corresponding transformation information. If the don't cares of f_c are implied by t, this indicates that G' can achieve fewer nodes or a reduced logic level. In this case, $\mathcal{C}$ is replaced by G'. The replacement is performed by applying $\mathcal{C}$'s previously computed permutation and negation vectors, adjusted to align with the G''s $\mathcal{NPN}$ class representative. This approach ensures that the replacement process considers both the functional equivalence and the optimization potential provided by the don't cares, thereby improving the overall quality of the circuit.

E. Workflow

The preceding sections have detailed the principal phases involved in DCLOG. Algorithm 1 summarizes the complete algorithmic procedure of our don't cares-based logic optimization using pre-training graph neural networks. The procedure systematically integrates incremental simulation and cut filtering to achieve efficient logic optimization with don't cares.

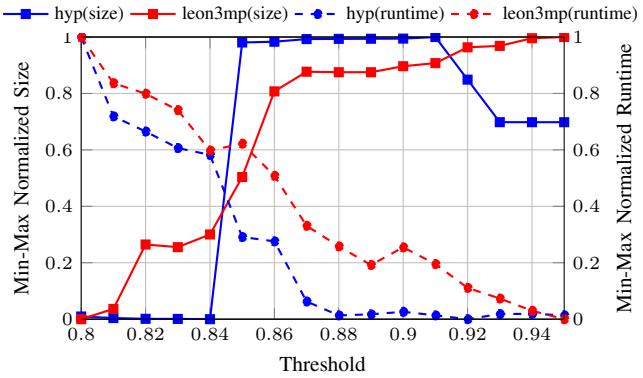

Fig. 3. Performance trade-off between size reduction and runtime efficiency across different optimization thresholds.

Initially, the output probability of each node in the Boolean network is predicted using the pre-trained model described in Section IV-C (line 1), enabling rapid estimation of node behaviors required for subsequent filtering. The procedure then iterates over all nodes in topological order from primary inputs to primary outputs (lines 2-22), thereby preserving dependency constraints and facilitating incremental updates.

For each node v, the procedure first calculates its current level (line 3) and identifies if it is on the critical path (line 4) to guide subsequent cut rewriting. An L-input reconvergent window is constructed for node v (line 5) to localize the calculation of don't cares. The K-cuts of node v are then enumerated (lines 7-19) to identify potential candidates for rewriting.

For each K-cut $\mathcal{C}$ of node v, the cut-filtering method described in Section IV-B is applied to determine whether $\mathcal{C}$ contains don't cares (lines 9-12). If so, the window-based don't care calculation method from Section IV-A is employed. Subsequently, Boolean matching is performed in the exact database as described in Section IV-D to identify alternative structures Gs (line 13).

For each candidate structure $G' \in Gs$, the gain cur from replacing $\mathcal{C}$ with G' is calculated (line 15). The gain includes two parts: {the level of node v after replacement l, the number of reduced nodes n}. The replacement selection considers three optimization criteria: ❶ node reduction under critical path delay improvement (line 16), ❷ node reduction on non-critical paths (line 17), and ❸ level reduction with the same number of nodes reduced (line 18). If any criterion is satisfied, the best candidate and its gain are updated (lines 16-19).

If an improved candidate is found, the replacement is applied to the network (line 21), and the output probabilities of newly added nodes are updated (line 22). The algorithm terminates after processing all nodes, returning the optimized network (line 23).

V. Experimental Results

The proposed don't cares-based logic rewriting framework was implemented using Python and C++. The prediction model is developed by the PyTorch deep learning framework. Besides, DCLOG employed the mockturtle [18], an open-source logic network library, to construct the exact library and implement Boolean matching with don't cares. For the evaluation, considering the efficiency and feasibility of logic rewriting, we used large circuits from the EPFL [19] and IWLS 2005 [20] benchmarks. Given that MIG has a more complex function and can support a more compact representation over other Boolean

TABLE I. The experimental results on MIG optimization.

Circuit	Original		Resubstitution [8]		Rewrite [5]		Ours		Runtime(s)		
	Size	Depth	Size	Depth	Size	Depth	Size	Depth	Resubstitution [8]	Rewrite [5]	Ours
div	57247	4372	53030	4338	46352	4307	**45922**	4366	**10.71**	28.66	24.55
hyp	214335	24801	199598	16717	156584	9154	**156182**	9155	**95.32**	147.94	138.24
arbiter	11839	87	**11711**	87	11839	87	11839	87	**1.18**	2.98	1.80
ethernet	86726	32	85976	31	66385	31	**61306**	34	18.91	33.12	**14.83**
leon2	789647	58	786502	57	**779285**	62	780385	64	3445.07	381.51	**152.31**
leon3_opt	974977	54	970964	55	**952861**	48	955675	50	5677.65	568.25	**172.23**
leon3	1088122	59	1083566	58	1014081	53	**1002184**	56	7010.14	782.32	**410.01**
leon3mp	652353	55	649958	47	590140	44	**590002**	48	1976.38	413.64	**208.40**
netcard	803848	40	801960	37	533652	36	**525241**	37	3007.48	773.25	**548.87**
pci_bridge32	22806	30	22461	28	17345	33	**16524**	33	**1.96**	7.02	2.47
vga_lcd	126708	24	126474	24	**89375**	24	89472	24	38.89	55.92	**25.07**
Ave. ratio	1	1	0.9821	0.9393	0.8397	**0.9072**	**0.8287**	0.9370	**8.4587**	**2.0277**	1

logic structures, we selected MIG for the logic rewriting task. For parameter configuration, we set the cut size $K = 4$ and the window size $L = 12$. The experiments were conducted on a machine running Ubuntu 22.04 and equipped with Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz, GeForce RTX 3090, and 256.0 GB of memory.

A. Node Output Prediction

As mentioned before, our node output probability predictor leverages a pre-trained FGNN2 model as the circuit encoder. This model then undergoes fine-tuning using the open-source DeepGate2 dataset, which is collected by performing MIG-based logic simulations with a large number of random patterns. The dataset is partitioned into training, validation, and testing sets in a ratio of 7:1:2. The fine-tuning process of the FGNN2 model entails training on the training dataset for 100 epochs, requiring approximately 1 hour to complete.

For the efficacy assessment of our node output probability predictor, we employ the R^2 score as the primary evaluation metric, widely recognized for regression tasks. Notably, our node output probability predictor achieves an impressive R^2 score of 0.995 on the testing dataset, affirming its accuracy and reliability. Thanks to the model's high prediction accuracy, we can filter out most cuts (up to 98.72%) lacking don't cares, while maintaining a low error rate (as low as 5.19%) in classifying cuts without don't cares as having them.

B. Rewriting with Don't Cares

We compared DCLOG against two baselines: ❶ the state-of-the-art logic resubstitution using don't cares [8], and ❷ the state-of-the-art exact logic rewriting with don't cares [5]. Both baselines were configured with the same parameters as DCLOG, and all results were validated for functional equivalence.

The selection of the threshold in our DCLOG framework is crucial for balancing runtime and optimization performance. Taking the circuits `hyp` and `leon3mp` as examples, we conducted experiments with various thresholds ranging from 0.80 to 0.95, as shown in Fig. 3. The results indicate that a threshold of 0.84 yields an effective trade-off between size reduction and runtime efficiency. Specifically, the size begins to increase sharply beyond the threshold of 0.84, while the runtime experiences a significant decrease. This suggests that a threshold of 0.84 is a reasonable choice for achieving a balance between size reduction and runtime efficiency. Furthermore, the precision of cut filtering under this threshold is 77.83% for the circuit `hyp` and 67.76% for the circuit `leon3mp`.

TABLE I presents the experimental results on MIG optimization. The "Original" part shows the initial attributes of input circuits, including "Size", the original number of majority nodes in the circuit, and "Depth", the original circuit depth. The "Resubstitution" and "Rewrite" parts represent the results of two baselines, respectively. The "Ours" parts present the results of DCLOG with a filter threshold $\alpha = 0.84$. In these three parts, "Size" and "Depth" indicate the node size and circuit depth after optimization. Furthermore, the "Runtime(s)" part shows the corresponding runtime in seconds of each method. For DCLOG, the runtime also includes circuit representation and model inference.

Experimental results demonstrate the effectiveness and efficiency of DCLOG, which reduces the size by 17.13% and the depth by 6.30% on average compared to the original circuit. When compared with the state-of-the-art logic resubstitution method [8], DCLOG achieves an average runtime reduction of 23.84% and an average size reduction of 15.64%. In comparison with the state-of-the-art exact logic rewriting method [5], DCLOG achieves an average runtime reduction of 44.70% while also reducing size by an average of 1.44%.

Overall, the experimental results demonstrate that DCLOG can effectively optimize the circuit size and depth while maintaining a low runtime. Since our DCLOG requires additional time for the circuit representation and model inference, it does not show an obvious advantage in runtime for small circuits compared with the state-of-the-art logic resubstitution method [8]. However, for large circuits, DCLOG can achieve remarkable efficiency, delivering over a 10x speedup. Moreover, DCLOG consistently requires less runtime than the state-of-the-art exact logic rewriting method [5] across all test cases, while also achieving a better size reduction.

VI. Conclusion

This paper presented DCLOG, an innovative framework that efficiently integrates don't cares into logic rewriting by leveraging a pre-trained graph neural network model. Specifically, we proposed a cut-filtering method to effectively discard cuts without don't cares and incorporated incremental simulation to reduce the computational costs of don't cares. Experimental results on large Boolean networks for MIG optimization demonstrated that DCLOG achieved significant computational efficiency improvements of 23.84% and 44.70% in runtime while substantially reducing circuit size by an average of 15.64% and 1.44%, compared to state-of-the-art logic resubstitution and rewriting methods with don't cares, respectively.

Acknowledgment

This work was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust, and was supported in part by the Research Grants Council of Hong Kong SAR (Grant No. CUHK14207523), and in part by the National Natural Science Foundation of China (Grant No. 62302477).

REFERENCES

[1] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[2] R. Fu, R. Zhang, Z. Zheng, Z. Shi, Y. Pu, J. Huang, Q. Xu, and T.-Y. Ho, "Late breaking results: Hybrid logic optimization with predictive self-supervision," in *ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–2.

[3] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 532–535.

[4] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2022, pp. 410–416.

[5] A. T. Calvino and G. De Micheli, "Scalable logic rewriting using don't cares," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2024, pp. 1–6.

[6] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 4, pp. 871–884, 2020.

[7] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 7, no. 6, pp. 723–740, 1988.

[8] S.-Y. Lee and G. D. Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 11, pp. 3958–3971, 2023.

[9] L. Benini and G. De Micheli, "A survey of boolean matching techniques for library binding," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 2, no. 3, p. 193–226, 1997.

[10] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 5, pp. 806–819, 2016.

[11] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, B. Yu, and Y. Huang, "Functionality matters in netlist representation learning," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, p. 61–66.

[12] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, Y. Huang, and B. Yu, "FGNN2: A powerful pretraining framework for learning the logic functionality of circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 44, no. 1, pp. 227–240, 2025.

[13] M. Li, S. Khan, Z. Shi, N. Wang, H. Yu, and Q. Xu, "Deepgate: Learning neural representations of logic gates," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, p. 667–672.

[14] Z. Shi, H. Pan, S. Khan, M. Li, Y. Liu, J. Huang, H.-L. Zhen, M. Yuan, Z. Chu, and Q. Xu, "DeepGate2: Functionality-aware circuit representation learning," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9.

[15] H. Riener, S.-Y. Lee, A. Mishchenko, and G. De Micheli, "Boolean rewriting strikes back: Reconvergence-driven windowing meets resynthesis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2022, pp. 395–402.

[16] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *IEEE/ACM International Workshop on Logic Synthesis*, 2006.

[17] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *International Conference on Machine Learning (ICML)*, 2013.

[18] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. Tempia Calvino, and G. Marakkalage, Dewmini Sudara De Micheli, "The EPFL logic synthesis libraries," 2022, arXiv:1805.05121v3.

[19] L. G. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *IEEE/ACM International Workshop on Logic Synthesis*, 2015.

[20] C. Albrecht, "IWLS 2005 benchmarks," Cadence Research Laboratories at Berkeley, Tech. Rep., Jun. 2005.