CircuitFlow: Learning Dynamic Representations for Logic Optimization

Miao Liu, Xinhua Lai, Liwei Ni, Xingyu Meng, Rui Wang, Junfeng Liu, Xingquan Li, Jungang Xu

***Abstract*—Graph neural networks have emerged as a powerful tool for learning circuit representation but largely ignore the dynamic nature of logic optimization (LO), leading to a severe "domain gap" where embedding quality degrades after heuristic transformations. To address this, we propose CircuitFlow, a framework learning representations for both circuits and optimization algorithms. CircuitFlow is distinguished by three features: (1) it employs circuit motifs and DAGTrans for robust encoding; (2) it utilizes a dual-component architecture where OPTrans models optimization operations via Transformers; and (3) it seamlessly integrates into LO pipelines to predict optimized circuit statistics.Experimental results demonstrate that CircuitFlow reduces physical metric prediction loss by up to 72% compared to baselines. Crucially, it effectively bridges the domain gap, bringing the generalization error on optimized circuits down from over 65% to within 20%. Furthermore, integrating CircuitFlow into the LO flow yields an average QoR improvement of 3.97% alongside a 3.11× speedup. Finally, we leverage CircuitFlow to introduce *orientational logic optimization*, a new paradigm guiding synthesis toward target structures, which reduces the runtime of Combinational Equivalence Checking (CEC) by 76% on average.**

***Index Terms*—EDA, logic optimization, representation learning, Transformer,**

I. Introduction

In chip design, Logic Optimization (LO) is a pivotal step for refining circuit structures to enhance Power, Performance, and Area (PPA). As the final quality of a chip heavily relies on LO, developing smarter and more efficient optimization methods remains a primary challenge. Recently, deep learning has emerged as a transformative force in this domain. Graph Neural Networks (GNNs), with their native ability to process graph-structured data, have become the dominant architecture, successfully applied to tasks such as Boolean satisfiability [1], [2], circuit classification [3], and testability analysis [4], [5].

Manuscript received 15 October 2025; revised 13 January 2026; accepted 13 Febrary 2026. This article was recommended by Associate Editor Hao Zheng. *(Corresponding authors: Jugang Xu, Email: xujg@ucas.ac.cn; Xingquan Li, Email: fzulxq@gmail.coms)*

This work was supported in part by the Major Key Project of PCL (No. PCL2025A04), and the NSF of Fujian Province under Grants (No. 2024J09045).The authors acknowledge that no generative AI technologies were used in the drafting or creation of this manuscript.

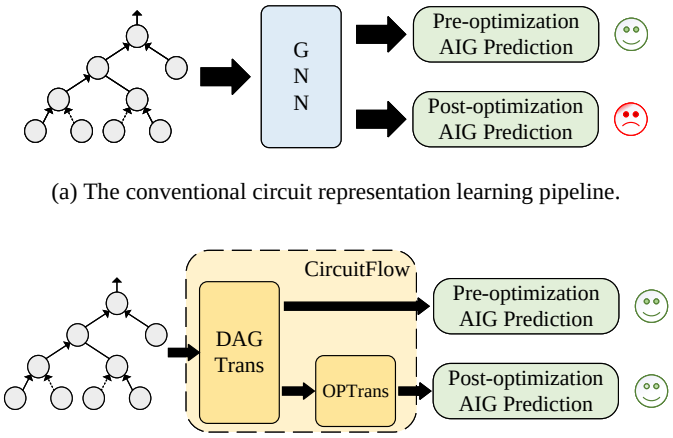
Miao Liu, Xinhua Lai, Jungang Xu are with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, 100049, China.

Liwei Ni is with Shanghai UniVista Industrial Software Group Co., Ltd., Shanghai, 200120, China.

Xingyu Meng, Junfeng Liu, Xingquan Li are with Pengcheng Laboratory, Shenzhen, 518055, China.

Rui Wang is with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, 518060, China.

Digital Object Identifier xx.xxxx/TCAD.xxxx.xxxxxxx.



(a) The conventional circuit representation learning pipeline.

(b) The CircuitFlow representation learning of circuit and algorithm.

Fig. 1. The "domain gap" challenge for GNNs in logic optimization and CircuitFlow's solution.

Building on this progress, pioneering works have integrated GNNs directly into synthesis flows. For instance, LSOformer [6] and MILS [7] utilize advanced architectures to model optimization sequences. Similarly, approaches like OpenABC-D [8] and RL-based methods [9], [10] employ GNNs to encode circuit states for Quality of Result (QoR) prediction or policy learning. While effective on established benchmarks, these methods exhibit two critical limitations when applied to the dynamic nature of LO:

- **Insufficient representation capability:** Standard GNNs fail to capture the Boolean functionalities and high-order structural patterns of logic circuits. Although functional-aware methods like DeepGate2 [1] and others [3], [11], [12] attempt to mitigate this via truth table supervision or specialized embeddings, they trade structural fidelity for functional accuracy, limiting their depth.
- **Domain gap in predictive tasks:** A significant semantic chasm exists between the input representation and the output prediction. Existing approaches typically learn a static embedding from a *pre-optimization* circuit but are tasked with predicting a *post-optimization* outcome [8]. As illustrated in Fig. 1, the GNN remains blind to the drastic, non-linear structural transformations occurring during LO, such as *rewrite*, *balance*, and *refactor*. This disconnect results in a severe performance drop when generalizing to optimized circuits.

However, there is little research to address the second limitation. While efforts to improve circuit representation have advanced on the functional front, they often do so at the

expense of structural fidelity and, more importantly, they do not address the fundamental domain gap inherent in dynamic optimization tasks. The core challenge, therefore, is not merely to represent a single circuit state with higher accuracy, but to understand and model the evolutionary trajectory of a circuit when it undergoes a sequence of complex transformations. An effective solution must not be blind to the optimization process but rather embrace it as a core component of the problem.

The core challenge, therefore, is not merely representing a static circuit state, but modeling its evolutionary trajectory under complex transformations. To address this, we propose **CircuitFlow**, a novel framework designed to bridge the domain gap in dynamic logic optimization. CircuitFlow employs a dual-component architecture:

- **DAGTrans:** A specialized graph Transformer that transcends the local receptive fields of conventional GNNs. By leveraging motif-based encodings and leaf aggregation, it captures high-order structural patterns and long-range dependencies for robust static embedding.
- **OPTrans:** A dynamic modeling Transformer trained to approximate the effects of optimization algorithms directly within the embedding space. It learns a nonlinear mapping from an initial state to a predicted post-optimization state.

By jointly learning from the circuit's state and the optimization actions, CircuitFlow effectively bridges the semantic border between states. Our primary contributions are summarized as follows:

- We propose CircuitFlow, a unified framework combining the static representation power of DAGTrans with the dynamic simulation capability of OPTrans.
- Experimental results demonstrate that CircuitFlow achieves over a 2.3× accuracy improvement in gate-level prediction and 2× in QoR prediction compared to baselines, achieving near-oracle fidelity in post-optimization metrics.
- Integrated into logic synthesis flows, CircuitFlow yields superior downstream performance, achieving an average QoR improvement of 3.97% with a 3.11× acceleration in DRiLLS [13], and a 7.11% improvement in CBtune [14].
- We introduce Orientational Optimization, a new paradigm enabled by our framework, which reduces the runtime by 76% on average for Combinational Equivalence Checking benchmarks.

The remainder of this paper is organized as follows: Section II covers preliminaries; Section III outlines the problem formulation; Section IV and Section V detail the data processing and model architecture; Section VI presents experimental results; Section VII explores applications; and Section VIII concludes the work.

II. Preliminaries

A. Logic Optimization Algorithms

A logic circuit can be formally represented as a DAG, denoted as $G = (V, E)$, where V is the set of nodes and E is the set of directed edges representing signal flow. In a common representation like an AIG, the nodes $v \in V$ correspond to Primary Inputs (PIs), Primary Outputs (POs), and two-input AND gates. Inverters are implicitly represented by attributes on the edges.

LO transforms an initial circuit G_0 into more optimal circuit G_t with equivalent functionalities. This is achieved by applying a sequence of transformation algorithms, referred as "recipes". Let $\mathcal{O} = \{o_1, o_2, \cdots, o_m\}$ be a predefined set of available optimization operations, such as *rewrite*, *balance*, and *refactor* [15], [16]. An optimization recipe is a sequence of these operations, $\mathcal{S} = \{s_1, s_2, \cdots, s_t\}$, where each $s_i \in \mathcal{O}$. The application of the recipe is a deterministic, sequential process by applying each operation in order. The goal of logic optimization is to find an optimal sequence S^* that minimizes a cost function, which is typically a weighted combination of area and delay or the product of them. Finding S^* is an NP-hard problem, and thus modern LO tools rely on expert-designed heuristics and scripts.

B. Circuit Representation Learning

To apply machine learning to circuit tasks, Circuit Representation Learning aims to learn an encoder mapping a circuit G to a d-dimensional vector. GNNs are the standard approach for this task, utilizing iterative message passing to update node features based on their neighbors. Recent architectures have significantly improved this standard process. For instance, Hoga [17] optimizes the aggregation via hop-wise attention to capture long-range dependencies, while PolarGate [12] and MILS [7] enrich node representations by integrating polar coordinate-based functional embeddings and mixed-level abstraction information, respectively. DynamicRTL [18] utilizes simulation traces to capture the temporal changes of signal values during execution. Finally, a readout function generates a graph-level embedding h_G, facilitating downstream tasks such as Boolean satisfiability [1], [2], circuit classification [11], and testability analysis [4], [5].

C. Algorithm Representation Learning

Beyond merely learning static circuit embeddings, representation learning can also be extended to model the dynamic impact of heuristic optimization algorithms. This paradigm, termed Algorithm Representation Learning, aims to capture the transformations applied in the feature space. Specifically, for a given heuristic algorithm A, we seek to learn a mapping function $f_A : \mathbb{R}^d \to \mathbb{R}^d$ that represents the changes made by heuristics on the embedding vector.

The Transformer model is well-suited for this task due to its inherent self-attention mechanism, which enables the model to dynamically weight the importance of different gates or subgraphs within the input embedding vector, simulating the focus of the optimization algorithm on the most salient features. Specifically, given the initial vector $h_g \in \mathbb{R}^d$, which is the circuit representation embedding, the output of the final Transformer layer is the transformed embedding vector $h_{A(g)} = f_A(h_g)$. This new vector $h_{A(g)}$ represents the circuit state after the application of heuristic algorithm. Similarly, these embeddings, representing the circuit, can also be the input for the downstream tasks.

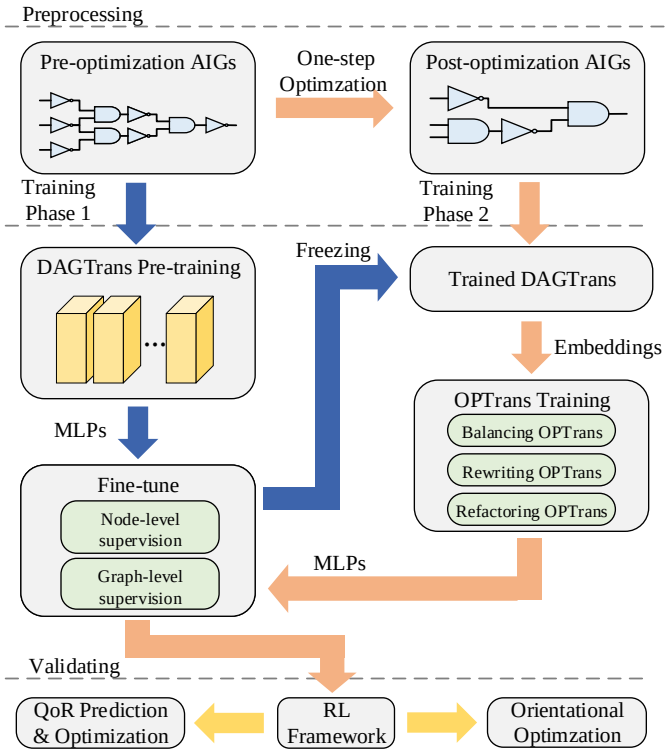


Fig. 2. The overall framework of CircuitFlow. The DAGTrans and MLPs are frozen during the training phase 2.

III. Problem Formulation and Framework Overview

This section formalizes the dual challenges of representing static circuits and dynamic optimization algorithms, followed by a concise overview of the CircuitFlow framework.

*Problem 1 (***Circuit Representation Learning):** Learn a mapping function $\Phi : \mathcal{G} \to \mathbb{R}^d$ that encodes a logic circuit G into a low-dimensional feature vector. This representation must capture both functional semantics and structural topology to facilitate downstream tasks such as component identification and property prediction.

*Problem 2 (***Algorithm Representation Learning):** Learn a transition function $\Psi : (\mathbb{R}^d, \mathcal{A}) \to \mathbb{R}^d$ that models the effect of an optimization operator $a \in \mathcal{A}$ in the feature space. The objective is to ensure that $\Psi(\Phi(G), a) \approx \Phi(G')$, where G' is the circuit obtained by applying a to G.

As illustrated in Fig. 2, CircuitFlow addresses these problems through a three-stage pipeline:

- **Preprocessing:** We generate a comprehensive dataset of AIGs and their optimized variants using ABC operators (e.g., *rewrite*, *balance*, *refactor*). This establishes ground-truth pairs for learning structural evolution and records corresponding QoR labels.
- **Training:** We employ a two-phase strategy. Phase 1 utilizes **DAGTrans** to extract static functional and structural features via leaf aggregation mechanisms. Phase 2 trains **OPTrans** to model optimization operators as learnable transformations, enabling the simulation of circuit evolution in the latent space.
- **Validation & Application:** The pre-trained models are fine-tuned and deployed for downstream tasks, including high-fidelity QoR prediction and the proposed orientational logic optimization (refer to Sections VI and VII).

IV. Circuit Information and Dataset

To facilitate robust representation learning, we extract two fundamental modalities from logic circuits: scalable functional signatures via simulation and high-order structural patterns via motifs.

A. Circuit Function Capture

The truth table serves as the canonical representation of circuit functionality, providing an exhaustive mapping from all possible input combinations to their corresponding outputs. Formally, for a combinational circuit with n inputs and m outputs, its functionality is precisely defined by a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$. However, the applicability of full truth tables is severely constrained by their exponential complexity. As the state space scales as 2^n, generating or processing these tables becomes computationally intractable for circuits with even a moderate number of PIs (e.g., $n > 20$). To circumvent this scalability bottleneck, existing methods typically restrict training to circuits with narrow input widths (e.g., $n < 12$) [1], [11], or resort to partitioning the circuit into smaller subgraphs for both GNN processing and label generation [2], [19].

To ensure scalability, we designate the logic-1 probability as the supervision target. For a node v, $P(v)$ is defined as the likelihood that its output evaluates to *True* under a stochastic distribution of inputs. Estimated via Monte Carlo simulation, this metric remains computationally tractable even for high-dimensional input spaces. Crucially, it distills high-dimensional functional semantics into a differentiable continuous scalar. By regressing these probabilities, the model enforces an embedding space where functional similarity translates into geometric proximity.

B. Circuit Structure Capture

While circuits are modeled as DAGs, standard GNNs are inherently limited by their local receptive fields, often failing to distinguish complex topological dependencies. To mitigate this, we enrich the representation by explicitly encoding circuit motifs, which are small, recurring subgraph patterns that represent fundamental building blocks of logic circuits.

Formally, we define a library of K motifs, $\mathbb{M} = \{M_1, \ldots, M_K\}$, as shown in Fig. 3. For each node $v \in V$, we compute a feature vector $x_v^{motif} \in \mathbb{N}^K$, where the i-th entry records the frequency with which node v participates in motif M_i. This encoding yields three critical advantages: (1) **Scalability**: It offers a computationally efficient alternative to NP-hard metrics like Graph Edit Distance, enabling large-scale circuit comparison; (2) **Global Perception**: It explicitly captures high-order structural patterns, transcending the locality of standard GNNs; (3) **Contextual Differentiation**: It enables the model to distinguish identical gate types based on their

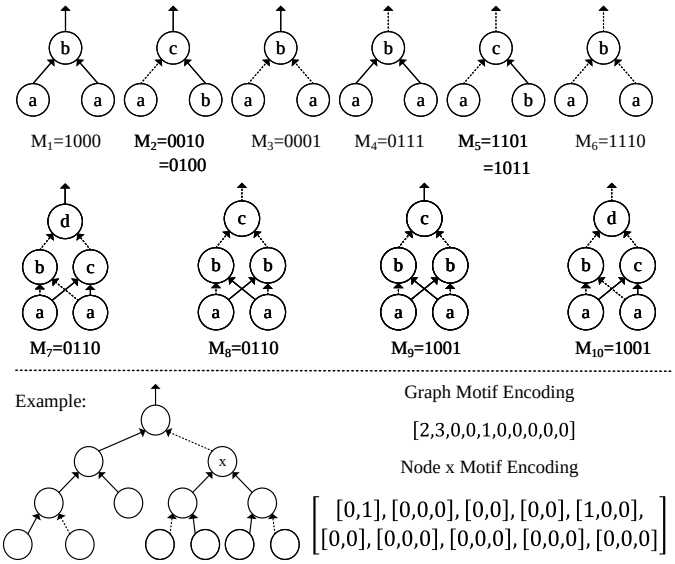


Fig. 3. All 12 2-input motifs that cover some combinations of logical truth-table, except 1111, 1100, 1010, and their reverses. M_2 and M_5 have 2 truth-tables. M_7 and M_8 share the same truth-table but different structures.

structural context (e.g., differentiating an AND gate in an XOR structure from one in an XNOR).

The encoding pipeline proceeds in two steps: (1) **Motif Selection**, where a set of basic subgraphs is curated based on their functional significance; and (2) **Isomorphic Matching**, where we compute the distribution of these motifs across the circuit, accounting for structural symmetries. For instance, if a node v maps to role a in motif M_1 once and role c in motif M_2 twice, its feature vector is populated as $[1, 0, \ldots, 2, \ldots, 0]$, where each index corresponds to a specific role within a specific motif. Notably, this counting process is efficient, with a time complexity of $\mathcal{O}(mn)$ [20], where m and n denote the sizes of the motif and the circuit, respectively.

To optimize this representation, we utilize a contrastive learning objective, ensuring that structurally isomorphic components are mapped to proximate locations in the embedding space. Crucially, this learned embedding aligns with the motif vector, enabling our OPTrans model to predict the structural composition of an optimized circuit directly from its initial state. By anchoring the representation in functionally significant subgraphs, we emulate the semantics of technology mapping, thereby empowering the GNN to capture robust, context-aware structural dependencies.

C. Dataset Preparation

To effectively train the proposed framework on both static circuit properties and dynamic optimization trajectories, we construct a comprehensive dataset designed to capture state transitions. The preparation pipeline consists of two phases: iterative data generation and multi-modal annotation.

1) Iterative Data Generation: To model the optimization process, we employ an iterative expansion strategy. For each source circuit G_i, we apply three distinct logic optimization algorithms to generate variants: $G_{rw} = \text{rewrite}(G_i)$, $G_b = \text{balance}(G_i)$, and $G_{rf} = \text{refactor}(G_i)$. This yields

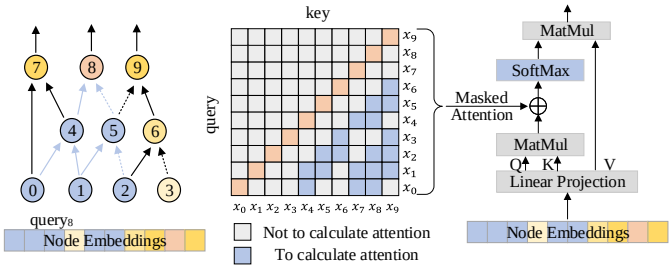


Fig. 4. Overview of leaf aggregation and leaf self-attention mechanism. Node 8 only considers the features from its leaf nodes (in blue) and aggregates the features in the self-attention mechanism.

paired examples (e.g., (G_i, G_{rw})) that serve as ground truth for learning operator-specific transformations. Crucially, the newly generated circuits are fed back into the source pool for subsequent rounds. We cap this recursive process at 3 iterations to mitigate exponential data growth, resulting in a dataset that is a four-fold expansion of the original benchmarks.

2) Comprehensive Annotation: Following generation, we perform an extensive annotation phase for every circuit graph. As established in preceding sections, we compute labels for both functional and structural modalities: specifically, the simulation-based logic-1 probability and the motif-based feature vector for each node. Additionally, we calculate similarity distances for graph motif features to support contrastive learning. Finally, to facilitate downstream QoR prediction, we perform technology mapping on all circuits to acquire ground-truth physical statistics (area and delay). Section V-C details the integration of these labels into our multi-task learning objective.

V. Transformer-based Model for Circuit Representation and Optimization Operators

This section presents CircuitFlow, implemented via a sequential, two-phase strategy. First, we train **DAGTrans** to encode static functional and structural semantics. Subsequently, utilizing DAGTrans as a frozen feature extractor, we train **OPTrans** to model the dynamic transformations induced by optimization algorithms. The specific architectures and learning objectives are detailed below.

A. *DAGTrans*

A primary limitation of conventional GNNs is their reliance on localized message passing, where the receptive field of a node expands by only one hop per layer. This constraint hinders the efficient capture of the long-range dependencies and hierarchical structures inherent in logic circuits. Inspired by recent works that leverage positional or level-based information to enrich graph representations [2], [17], we propose **DAGTrans**, a Transformer-based architecture specifically designed to directly capture non-local structural information and global circuit dependencies.

The cornerstone of DAGTrans is the leaf aggregation mechanism, as illustrated in Fig. 4. We formally define the leaves of a target node v, denoted $L(v)$, as the set of all nodes in its transitive fan-in cone. That is, a node u is in $L(v)$ if

and only if there exists a directed path from u to v. This set represents the entire logic passing history of node v. Its logical state is exclusively determined by the states of its leaves. Consequently, these nodes are more relevant than other nodes in the circuit both functionally and structurally.

The core operation of each layer is to update a node's embedding by aggregating information directly from its leaves, rather than just its immediate parents. The aggregation at layer k is defined as :

$$h_{L(v)}^{k} = \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in L(v)\}) \tag{1}$$

where h_u^{k-1} is the embedding of a leaf node u from the previous layer. Because the leaves $L(v)$ naturally span multiple levels of logic depth within the circuit, this single aggregation step can capture high-order structural information without the need for many message-passing iterations.

In order to implement a powerful AGGREGATE function, we employ a self-attention mechanism, which allows the model to dynamically weight the importance of each leaf node when updating the target node v. Unlike simple pooling operations, self-attention aggregation enables the model to selectively focus on the most relevant parts of the fan-in cone, learning a context-dependent aggregation strategy. The attention-based aggregation is formulated as:

$$Attention(h_v) = \sum_{u \in L(v)} \frac{\kappa(h_v, h_u)}{\sum_{w \in L(v)} \kappa(h_v, h_w)} f(h_u). \tag{2}$$

$$\text{where} \quad f(x) = \mathbf{W_V} \cdot x, \tag{3}$$

$$\text{and} \quad \kappa(x, y) = exp\left(\frac{\langle x\mathbf{W_Q}, y\mathbf{W_K}\rangle}{\sqrt{d_\mathbf{K}}}\right) \tag{4}$$

where $\mathbf{W_Q}, \mathbf{W_K}, \mathbf{W_V}$ are trainable weight matrices that project the embeddings into Query, Key, and Value spaces, respectively. The attention score $\kappa(h_v, h_u)$ quantifies the relevance of leaf node u (key) to the target node v (query).

By capturing features directly from all antecedent nodes, this mechanism can naturally model complex structural phenomena. Such as reconvergent fan-outs, where a node influences a downstream target via multiple paths, it will be implicitly captured by different weights of the attention parameters. The self-attention mechanism can learn to assign a higher weight to such a fan-out node, recognizing its heightened influence as a shared logic. This ability to generate structurally-aware embeddings is not only powerful for static analysis but also provides a robust foundation for the subsequent OPTrans to learn the effects of circuit transformations.

B. *OPTrans*

The primary objective of the Optimization Transformation models (OPTrans) is to learn a set of non-linear transformations in the embedding space, where each transformation models the effect of a specific logic optimization algorithm. Formally, for an optimization algorithm *op* and any given circuit G, we seek to learn a function $F_{op} : \mathbb{R}^d \to \mathbb{R}^d$, such that $F_{op}(h_G) \approx h_{op(G)}$, where h is the embedding produced by DAGTrans. Once trained, these models allow us to predict the properties of an optimized circuit directly from the initial circuit embedding.

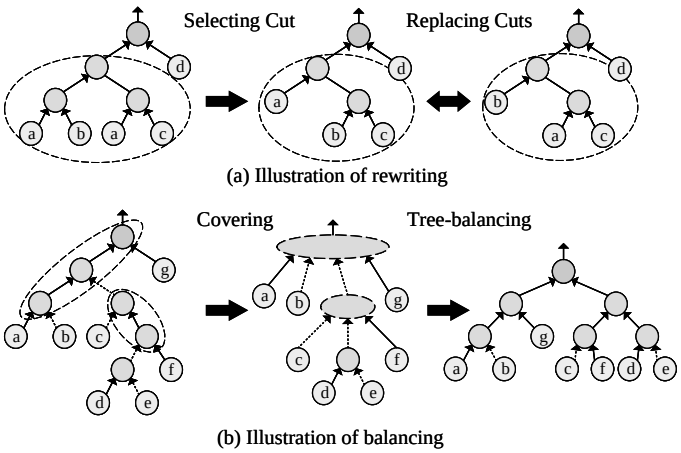


Fig. 5. The architecture of OPTrans is inspired from the optimization algorithm.

The training of each OPTrans model is conducted after the DAGTrans model has been fully trained and its parameters are frozen, ensuring that OPTrans learns on a stable representation space. A core principle of our design is that the architecture of each OPTrans must be tailored to the intrinsic properties of the algorithm it models. To achieve this, we propose specialized architectures for these fundamental optimization algorithms.

1) Rewriting OPTrans: Rewriting is a localized and stochastic process, as shown in Fig. 5(a). It iteratively identifies small k-feasible subgraphs (cuts) rooted at a node and replaces them with functionally equivalent substitutions found in a pre-computed library. This implies two key properties: (1) the transformation is local, affecting only a small subset of nodes, and (2) the structural context and topological order remain unchanged for the majority of nodes in the circuit.

To precisely model this behavior, Rewriting OPTrans, shown in Fig. 6, receives the node embeddings from DAGTrans as its primary input. To accommodate potential node generation, we employ a Latent Expansion Buffer by concatenating zero vectors to the input sequence. Furthermore, to retain the topological context of nodes within the DAG, we explicitly add sinusoidal positional encodings to the embeddings, ensuring that structural order is preserved during batch processing. It serves strictly as an implementation convention to provide unique structural identifiers for the permutation-invariant Transformer, rather than implying an inherent linear semantic order within the DAG.

Reserved Space Allocation Strategy. We empirically set the size of this reserved buffer to 20% of the input node count. This configuration is based on two key considerations:

- *Sufficiency vs. Noise:* Allocating excessive space introduces "padding noise" that dilutes valid attention signals, while insufficient space risks truncation. A 20% margin empirically strikes an optimal balance, providing sufficient capacity for localized node growth without compromising attention density.
- *Structural Placeholders:* Since OPTrans operates in a predictive mode targeting high-level metrics (e.g., Motif Counts) rather than exact netlist reconstruction, these reserved vectors act as "structural placeholders." They al-

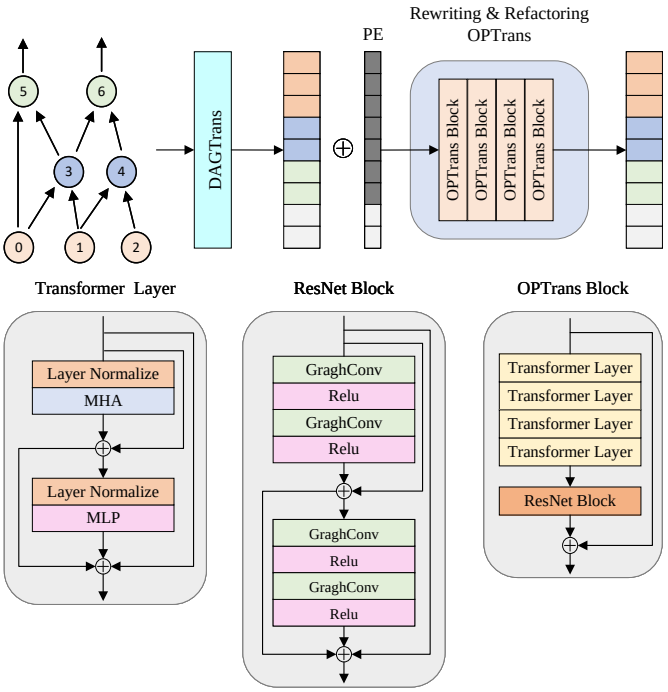


Fig. 6. The architecture of rewriting OPTrans and refactoring OPTrans.

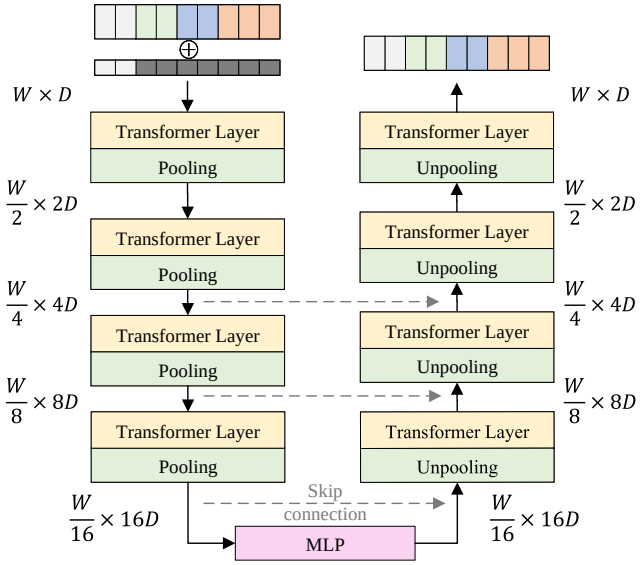


Fig. 7. The architecture of balancing OPTrans.

low the embedding space to represent increased structural complexity without requiring a strict one-to-one mapping to physical gates.

The model architecture is built around a main residual connection, which passes the initial embeddings directly to the final output. This preserves the representation of unchanged nodes, while the transformative layers learn to compute the updates. The core building block is repeated four times, including: (1) Four Transformer layers, each utilizing a multi-head attention mechanism to capture global dependencies, and (2) a ResNet block composed of graph convolutional layers with residual connections to refine local structural features.

By integrating a Transformer for global context, a GCN-based ResNet for local structure, and the latent expansion buffer, Rewriting OPTrans effectively captures the dynamics of the rewriting algorithm. The deep, multi-layered structure learns the complex transformation for the affected nodes, while the global residual connection preserves stability for the unaffected majority.

2) Balancing OPTrans: Balancing is a structurally predictable transformation aimed at reducing logic depth, as shown in Fig. 5(b). It proceeds in two main steps: first, it groups chains of two-input AND gates into larger, multi-input supergates (covering), and second, it decomposes these as balanced trees (tree-balancing). A key invariant in this process is that the root nodes of these logic cones preserve their functional role and connectivity to the rest of the circuit, while the internal nodes are systematically reorganized.

This structure, characterized by invariant roots and reconstructed internals, strongly motivates a U-Net-like architecture with hierarchical pooling and unpooling layers, inspired by Graph U-Net [21]. As illustrated in Fig. 7, the Balancing OPTrans features a multi-layer encoding path and a corresponding decoding path, connected by a central bottleneck and skip connections.

The encoding path comprises a series of blocks, each consisting of a Transformer layer followed by a pooling operation. The Transformer layer processes the node embeddings to capture global dependencies via self-attention, similar to the rewriting OPTrans. The subsequent pooling layer is designed to progressively downsample the graph embeddings, selecting and compressing the most salient features while reducing dimensionality. This process iteratively extracts higher-level, abstract representations as information flows deeper into the encoder.

At the bottleneck of the encoder, the most compressed representation is passed through an MLP, which acts as a projection mapping the invariant node features to their counterparts in the balanced circuit. Symmetrically, the decoding path consists of unpooling operations followed by Transformer layers. The unpooling layer is responsible for restoring dimensionality and propagating the learned changes back to individual node embeddings. Crucially, skip connections (dashed gray arrows in Fig. 7) are employed to directly transfer features from the encoding path to the corresponding levels of the decoding path. These connections are vital for preserving fine-grained details that might otherwise be lost during pooling, thereby allowing the decoder to reconstruct the transformed circuit embeddings with high fidelity.

Leveraging Transformer layers for global context and pooling/unpooling operations for multi-scale processing, the encoder-decoder architecture enables Balancing OPTrans to explicitly capture the structural equivalences inherent to the balancing algorithm. Consequently, it effectively predicts the resulting depth reduction and structural reorganization.

3) Refactoring OPTrans: Refactoring serves as a specialized variation of rewriting. While it shares the underlying mechanism of subgraph replacement, its primary objective is to mitigate logic reconvergence by identifying and synthesizing factored-form representations. Consequently, although

refactoring remains a localized transformation, it targets distinct structural outcomes compared to standard rewriting.

Given this operational similarity, we reuse the masked residual architecture for the Refactoring OPTrans. This strategy is both parameter-efficient and architecturally unified. Despite sharing the same framework, the model is trained with a distinct set of weights, enabling it to specialize in the nuances of refactoring. By doing so, the model learns a transformation function tailored to specific structural modifications—such as creating shared logic to reduce fan-out reconvergence—thereby distinguishing itself from pure size-minimizing rewriting.

4) Scalability Analysis: Although the current instantiation of OPTrans focuses on only three representative operators, the proposed Transformer-based architecture is designed to support the scalability requirements of modern logic synthesis tools, driven by its representation learning capabilities.

- Intra-operator (e.g., rewrite `-z`): The model supports parameter extensions via hierarchical prediction heads, leveraging the fine-grained structural features already captured by the encoder.
- Inter-operator (e.g., resubstitution): We clarify that OPTrans predicts latent feature representations rather than explicit structural modifications. Therefore, extending to other operators does not require a new modeling paradigm, but rather the construction of a tailored Transformer structure. By configuring the attention mechanism to aggregate global functional dependencies into the feature vectors, the framework can effectively simulate operators, decoupling the feature learning from the other specific Boolean transformations.

C. Training Tasks

To produce meaningful embeddings that capture both fine-grained gate properties and global circuit statistics, we employ a multi-task learning strategy. This process is divided into specific gate-level and graph-level objectives.

1) Gate Level Tasks: We enforce context-aware node representations through two mechanisms: functional prediction and structural contrast.

To capture the functional behavior of gates, we adopt the logic-1 probability as a supervision target. Estimated via extensive logic simulation, this probability serves as a continuous and differentiable proxy for the discrete truth table. By predicting this value, the model is compelled to capture the probabilistic behavior of the entire logic cone. We minimize the **Mean Absolute Error (MAE)** between the predicted probability and the simulated ground truth p_k:

$$L_{prob}^{gate} = \frac{1}{|V|} \sum_{k \in V} |p_k - \text{MLP}_{prob}(h_k)| \tag{5}$$

To encode structural similarity, we employ a contrastive learning mechanism. We define similarity based on the motif vectors encoded in Section IV. For any given anchor node v_a, we identify a positive sample v_p as a node with a highly similar motif vector (normalized Hamming distance $< \tau$), and a negative sample v_n as a node with a dissimilar vector. We minimize the triplet margin loss to pull similar nodes closer while pushing dissimilar ones apart:

$$L_{dist}^{gate} = \frac{1}{|\mathcal{T}|} \sum_{(a,p,n) \in \mathcal{T}} \max\left(0, d(h_a, h_p) - d(h_a, h_n) + \alpha\right) \tag{6}$$

where $d(\cdot)$ denotes the squared Euclidean distance and α is the margin hyperparameter.

2) Graph Level Tasks: A graph-level embedding h^{graph} is obtained via a sum readout to capture global circuit composition and physical metrics.

We predict the motif vector M_g with MSE loss, forcing the global embedding to retain information about circuit building blocks:

$$L_{motif}^{graph} = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} ||M_g - \text{MLP}_{motif}(h_g^{graph})||_2^2 \tag{7}$$

Crucially, to ground the abstract representation in pragmatic, physical-level outcomes, we perform regression on the post-synthesis Area and Delay using **Mean Squared Error (MSE)**:

$$L_{area}^{graph} = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} (Area_g - \text{MLP}_{area}(h_g^{graph}))^2 \tag{8}$$

$$L_{delay}^{graph} = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} (Delay_g - \text{MLP}_{delay}(h_g^{graph}))^2 \tag{9}$$

3) Two-Phase Optimization Strategy: Instead of optimizing all objectives simultaneously, we employ a curriculum learning strategy to establish a high-fidelity static space before learning dynamic evolution.

Phase 1: Static Representation Pre-training. We focus on training the DAGTrans encoder. The loss function is a weighted sum of the static tasks, ensuring precise magnitude calibration for physical properties:

$$\begin{aligned}\mathcal{L}_{\text{Phase1}} = \lambda_1 L_{gate}^{prob} + \lambda_2 L_{gate}^{dist} \\ + \lambda_3 L_{graph}^{motif} + \lambda_4 (L_{graph}^{area} + L_{graph}^{delay})\end{aligned} \tag{10}$$

Phase 2: Optimization Transformation Learning. We introduce OPTrans dynamics model while freezing the encoder. To capture evolution of circuits during optimization, we minimize the **Cosine Embedding Loss** between the predicted next-state embedding $\hat{h}t + 1$ and the ground truth $ht + 1$:

$$\hat{h}_{t+1} = T_{OPT}(h_t) \tag{11}$$

$$h_{t+1} = T_{DAG}(G_{t+1}) \tag{12}$$

$$\mathcal{L}_{\text{Phase2}} = 1 - \text{CosineSimilarity}(\hat{h}_{t+1}, h_{t+1}) \tag{13}$$

This decoupling ensures the embedding space is first geometrically structured to reflect physical metrics , and then trained to preserve semantic orientation during state transitions.

VI. Experiments

A. Preparation and Setup

- **Dataset and Tools:** We utilize the DeepGate dataset [25], comprising 10,824 AIG subcircuits. The dataset is strictly partitioned into training, validation, and testing sets with an 8:1:1 ratio. The ground-truth labels for optimized

TABLE I
LOSS COMPARISON WITH DIFFERENT REPRESENTATION MODELS FOR TASKS ON TEST DATASET

Loss \ Model	GraphSAGE [22]	DeepGate2 [1]	DeepGate3 [2]	PolorGate [12]	HOGA [17]	DAGformer [23]	GraphFormer [24]	Our DAGTrans
L_{prob}	0.3072	0.0225	0.0159	0.0276	0.157	0.1572	0.2189	**0.0102**
L_{dist}	1.714	0.8598	0.8163	0.9122	1.3128	1.211	1.598	**0.6316**
L_{motif}	1.533	0.8352	0.6924	0.8965	1.4980	0.6309	0.9553	**0.5013**
L_{area}	0.9836	0.2826	0.1594	0.1662	1.16	0.4325	0.5331	**0.0836**
L_{delay}	0.9861	0.7892	0.2291	0.3427	4.9061	0.925	0.6775	**0.2095**

TABLE II
PERFORMANCE COMPARISON OF DIFFERENT OPTIMIZATION ALGORITHMS FOR GRAPH-LEVEL PREDICTION

Optimization Algorithm	MSE	GraphSAGE		DeepGate2		DeepGate3		PolarGate		CircuitFlow	
		Loss	Reduction	Loss	Reduction	Loss	Reduction	Loss	Reduction	Loss	Reduction
Balancing	Motif	1.900	20.33%	1.032	26.08%	0.7908	14.21%	1.1209	25.03%	**0.6248**	19.83%
	Area	1.288	30.95%	0.4836	71.13%	0.2376	49.05%	0.2780	67.28%	**0.0995**	19.01%
	Delay	1.502	52.32%	1.020	29.24%	0.3117	36.05%	0.5076	48.11%	**0.255**	20.95%
Rewriting	Motif	1.960	24.13%	1.133	38.42%	0.8116	17.22%	1.0715	19.52%	**0.6116**	17.29%
	Area	1.350	37.25%	0.4690	65.96%	0.2394	50.19%	0.2827	70.12%	**0.1078**	28.94%
	Delay	1.726	75.03%	0.9663	22.44%	0.2933	28.02%	0.5354	56.23%	**0.2064**	24.29%
Refactoring	Motif	2.086	32.11%	0.9461	15.58%	0.7965	15.03%	1.1055	23.31%	**0.5583**	7.07%
	Area	1.211	23.12%	0.4665	65.07%	0.249	56.21%	0.2679	61.19%	**0.0988**	18.18%
	Delay	1.965	99.27%	1.041	31.91%	**0.2702**	17.94%	0.5215	52.18%	0.2850	24.18%

circuits are generated using the logic synthesis tool ABC [26] with three optimization commands: *balance*, *rewrite*, and *refactor*. To evaluate the QoR, we map the circuits using the `asap7.lib` [27] technology library.

- **Implementation Details:** Our proposed framework, CircuitFlow, is implemented using **`PyTorch Geometric` (v2.0.2) with `PyTorch` (v1.9.1)**. (1) Architecture: For *DAGTrans*, we employ a single round of message passing with a hidden dimension of $d = 128$. The leaf-aggregation mechanism utilizes a multi-head attention module with 4 heads. For *OPTrans*, the Rewriting and Refactoring models consist of 4 OPTrans Blocks. The Transformer components are uniformly configured with 4 attention heads , a hidden dimension of 128, and a dropout rate of 0.1. The Balancing OPTrans utilizes a U-Net-like structure with 4 encoder layers and 4 decoder layers. (2) Prediction Heads: The downstream prediction heads for each task are MLPs, consisting of a Linear layer, a ReLU activation, another Linear layer, and a final Sigmoid function for normalization.
- **Baselines:** We compare our model against a comprehensive set of baselines: GraphSAGE [22] (implemented in Gamora [28]), DeepGate2 [1], DeepGate3 [2], PolarGate [12], HOGA [17], and GraphFormer [29]. To ensure a fair comparison, all baselines are configured with a hidden dimension of 128. For specific configurations:
  - **DeepGate2:** We utilize the *Stage-2* model.
  - **PolarGate:** We configure the model depth with $L = 9$.
  - **Others:** We follow the default architectures in their public codes.
- **Training Configuration:** All models are trained using the Adam optimizer [30] with a weight decay of 1×10^{-10} on a single NVIDIA V100 DGXS 32GB GPU. The training is conducted in two phases:
  - **Phase 1 :** We train for a maximum of 80 epochs with an early stopping patience of 20 epochs. The learning rate is set to 1×10^{-4}. The weight parameters λ are all set to 1.
  - **Phase 2 :** We train for a maximum of 200 epochs with an early stopping patience of 50 epochs. The learning rate is refined to 1×10^{-5}.

B. Evaluation on Different Tasks

The overall performance on test datasets is summarized in Table I. The results highlight three key advantages of DAGTrans:

1. Dominance in Physical Prediction: DAGTrans achieves an order-of-magnitude reduction in physical metric losses. Specifically, for L_{area}, our model reaches 0.0836, which is nearly half lower than the runner-up DeepGate3 (0.1594). Similarly, for L_{delay}, we outperform PolarGate (0.3427) with a loss of 0.2095. This suggests that our leaf-aggregation mechanism effectively synthesizes path-critical information, enabling precise regression of global QoR metrics.

2. Balanced Functional and Structural Fidelity: While DeepGate3 remains competitive in functional prediction (L_{prob} of 0.0159 vs. our 0.0102), it struggles with structural fidelity (L_{dist} of 0.8163 vs. our 0.6316). This indicates that while simulation-guided baselines capture "what the circuit does," they miss "how it is built." DAGTrans bridges this gap by explicitly encoding motifs, ensuring both logic and topological patterns are preserved.

3. Inadequacy of Generic GNNs and Graph Transformers: General-purpose models, ranging from GNNs like GraphSAGE to Transformers like GraphFormer, exhibit substantially higher losses across tasks (e.g., GraphSAGE $L_{dist} = 1.714$, GraphFormer $L_{dist} = 1.598$). This confirms that neither isotropic message passing nor standard global attention mechanisms can effectively capture the directed logic flows inherent in circuits, validating the need for domain-specific architecture.

C. Evaluation on Different Optimization Algorithms

To evaluate the model's performance under distribution shifts, we test graph-level prediction accuracy on circuits transformed by *Balancing*, *Rewriting*, and *Refactoring*. The results in Table II provide two critical insights regarding generalization and domain adaptation.

1. Superior Generalization Capability: The MSE metric directly reflects the model's ability to generalize to "out-of-distribution" (OOD) samples generated by optimization operators. *Direct Analysis:* As the circuit structures change significantly after optimization, baseline models fail to generalize. For instance, DeepGate2's Area MSE degrades to ≈ 0.48, and Delay MSE exceeds 1.0. GraphSAGE performs even worse with losses over 1.2. *CircuitFlow Performance:* In contrast, CircuitFlow demonstrates exceptional generalization. Despite the structural shifts, it maintains a low Area MSE of ≈ 0.10 and Delay MSE of ≈ 0.25 across all algorithms. Where DeepGate3 achieves a comparable Delay loss (0.2702) to CircuitFlow (0.2850), our model consistently dominates in all other metrics, proving its robust generalization capability on unseen structural patterns.

2. Mitigation of the Domain Gap: The "Reduction" column quantifies the performance degradation (percentage increase in loss) on optimized circuits compared to the general test set results in Table I. This comparison explicitly reveals the magnitude of the **Domain Gap**. However, the magnitude of this error indicates how effectively a model bridges this gap. Baselines succumb to the domain gap, with DeepGate2 exhibiting Area reduction errors as high as 71.13% (Balancing) and 65.96% (Rewriting), rendering them unreliable for guiding optimization. Conversely, CircuitFlow effectively minimizes the impact of the domain gap. Its reduction errors are consistently controlled between 7.07% and 28.94%. This significant reduction in error proves that CircuitFlow successfully adapts to the shifted domain, providing a stable and reliable signal for downstream optimization tasks.

D. Validation of CircuitFlow

To rigorously quantify the contribution of the OPTrans module, we conduct a comparative study across three experimental settings, effectively establishing a lower bound, an upper bound, and our model's performance.

- *DAGTrans Original (Zero-shot Baseline)*: We directly apply the DAGTrans model trained on *unoptimized* circuits to predict the QoR of *optimized* circuits. High loss here indicates a domain gap.
- *DAGTrans Optimized (Oracle Upper Bound)*: We re-train DAGTrans from scratch using the ground-truth *optimized* circuits. This represents the theoretical best performance achievable if the model had full access to the target geometry.
- *CircuitFlow (Proposed)*: We use the full framework: the pre-trained DAGTrans embeds the *unoptimized* circuit, and OPTrans predicts the metrics of the *optimized* state without seeing the optimized netlist.

Table III reveals three critical insights:

TABLE III
VALIDATION OF CIRCUITFLOW

Settings	MSE	Balancing	Rewriting	Refactoring
DAGTrans Original	Motif	0.9618	0.9262	1.026
	Area	0.723	0.8002	0.6756
	Delay	0.8845	0.7962	0.8419
DAGTrans Optimized	Motif	0.5091	0.5119	0.5180
	Area	0.0385	0.0367	0.0337
	Delay	0.1549	0.1686	0.1707
CircuitFlow	Motif	0.6248	0.6116	0.5583
	Area	0.0995	0.1078	0.0988
	Delay	0.2550	0.2604	0.2850

TABLE IV
ABLATION STUDY OF DAGTRANS

Loss \ Setting	DAGTrans	DAGTrans w/o leaf aggr.	DAGTrans w/o motif encoding
L_{prob}	0.0102	0.0334	0.0133
L_{dist}	0.6316	0.9615	1.022
L_{motif}	0.5214	0.7108	0.9856
L_{area}	0.0436	0.1546	0.1723
L_{delay}	0.2095	0.2278	0.4832

TABLE V
ABLATION STUDY OF OPTRANS

Settings	MSE	Balancing	Rewriting	Refactoring
Deepgate2 + Vanilla Transformers	Motif	1.315	1.598	1.469
	Area	0.7963	0.8327	0.8801
	Delay	1.513	2.082	1.7628
Deepgate2 + OPTrans	Motif	0.9240	1.137	1.205
	Area	0.5128	0.5324	0.6011
	Delay	0.9286	1.267	1.303
DAGTrans + Vanilla Transformers	Motif	0.6332	0.8923	0.9019
	Area	0.2124	0.6722	0.7031
	Delay	0.3392	0.7138	0.6533
CircuitFlow	Motif	0.6248	0.6116	0.5583
	Area	0.0995	0.1078	0.0988
	Delay	0.2550	0.2604	0.2850

1. Confirmation of Domain Gap: The high losses of *DAGTrans Original* (e.g., Delay consistently > 0.80) confirm that static encoders trained on unoptimized circuits fail to generalize to the shifted distribution of optimized variants.

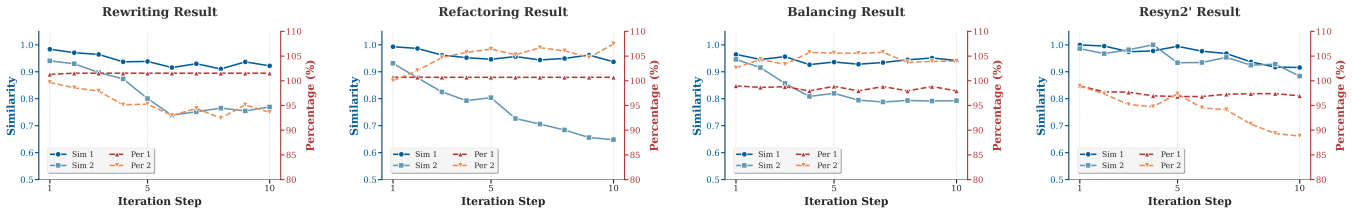
2. Effective Gap Bridging: *CircuitFlow* dramatically reduces these errors, proving its ability to learn structural transformations. For *Balancing*, the Area loss drops precipitously from 0.723 to 0.0995.

3. Near-Oracle Fidelity: Crucially, our predictive performance approaches that of the *DAGTrans Optimized* oracle (Area loss $\approx$ 0.037). This demonstrates that CircuitFlow recovers the majority of structural information, serving as a high-fidelity surrogate for expensive synthesis runs.

E. Ablation Study

1) Validation of DAGTrans: To validate the architectural contributions of DAGTrans, we conduct ablation studies on two key components: the leaf aggregation mechanism and the motif-based embedding. The quantitative results are presented in Table IV.

- *DAGTrans w/o leaf aggr.*: We replace our attention-based leaf aggregation with the standard predecessor



(a) (b) (c) (d)

Fig. 8. The accumulated error of CircuitFlow.

aggregation used in DAGNN [31], testing the importance of global context aggregation.

- *DAGTrans w/o motif encoding*: We remove the motif-based initialization and use *one-hot encoding of gate types* instead, evaluating the necessity of embedding higher-order topological information.

1. Impact of Leaf Aggregation: Removing this mechanism causes Area MSE to surge by 254.6% (to 0.1546), confirming its criticality for capturing global path dependencies required for physical metric estimation.

2. Impact of Motif Encoding: Replacing motifs with simple gate types increases structural distance loss by 61.8% and roughly doubles Delay MSE. This highlights that motifs are essential for embedding high-order functional and structural properties that one-hot encodings miss.

2) Validation of OPTrans: To validate the architectural design of OPTrans, we evaluate its effectiveness as a dynamic updater against a standard baseline across different backbone encoders. The results are summarized in Table V.

We employ a 2×2 factorial design comparing:

- **Backbones:** *DeepGate2* [1] (SOTA simulation-based encoder) vs. *DAGTrans* (Ours).
- **Updaters:** *Vanilla Transformer* vs. *OPTrans*. Note that the *Vanilla Transformer* is configured with identical hyperparameters (e.g., layers, heads, hidden dimensions) to the Transformer module within OPTrans, but it lacks the specialized hybrid components such as the residual blocks and latent expansion buffers.

1. Architectural Superiority: OPTrans consistently outperforms the Vanilla baseline. On the *DeepGate2* backbone, upgrading to OPTrans reduces Rewriting Delay loss by $\approx 39\%$ (from 2.082 to 1.267). Similarly, on *DAGTrans*, Rewriting Area loss drops dramatically from 0.6722 to 0.1078. This confirms that standard Transformers struggle with the "local-edit, global-impact" nature of logic optimization, whereas OPTrans's specialized hybrid architecture captures these dynamics effectively.

2. Model Generality: Crucially, OPTrans functions as a generic, plug-and-play dynamic head. Even with the fixed *DeepGate2* backbone, integrating OPTrans yields substantial gains (e.g., Balancing Area loss improves from 0.7963 to 0.5128). This demonstrates that OPTrans successfully extracts the *dynamic sensitivity* of circuits, effectively complementing the static features learned by various state-of-the-art encoders.

F. Analysis of Sequential Error Accumulation

To evaluate the stability of our model in recursive generation tasks, we conducted a long-horizon rollout experiment ($L = 10$) on the EPFL `log2` circuit. At each step, the model predicts the next-state embedding, which is then recursively used as input for the subsequent step. Fig. 8 visualizes the trajectories of both optimization performance and model fidelity through four distinct metrics:

- **Settings:** Configurations (a), (b), and (c) only employ a single *rewriting*, *refactoring*, or *balancing* operation, respectively. `resyn2'` is derived from `resyn2` by omitting the $-z$ parameter.
- **`per1`:** The *ground-truth* Area-Delay Product (ADP) at each step, simulated by ABC and normalized by the initial circuit's ADP.
- **`per2`:** The *predicted* normalized ADP, inferred recursively from the model's embeddings.
- **`sim1`:** The numerical ratio between the predicted QoR and the actual QoR.
- **`sim2`:** The Cosine Similarity between the predicted embedding (from OPTrans) and the ground-truth embedding (from DAGTrans).

As illustrated in Fig. 8, a divergence emerges between the two metrics during recursive rollout. The decay in embedding similarity (`sim2`) confirms that cumulative errors are present and inevitable over long horizons. However, the QoR prediction similarity (`sim1`) remains consistently high (> 0.9), yielding two key insights:

- **Robust Semantic Preservation:** The stability of `sim1` despite the drift in `sim2` suggests that OPTrans preserves underlying circuit semantics even when latent embeddings shift. The non-linear nature of the network effectively maps these "drifted" embeddings to accurate QoR estimates, bridging the structural-semantic gap.
- **Learning Sufficiency:** This sustained predictive accuracy under multi-step stress tests proves that CircuitFlow learns robust optimization trajectories rather than simple point-to-point mappings.

Therefore, the impact of cumulative error on downstream tasks is significantly mitigated, ensuring reliable QoR estimation for optimization sequences.

VII. Downstream Tasks

This section demonstrates CircuitFlow's utility and robustness in two critical logic synthesis tasks: optimization sequence generation and Combinational Equivalence Checking (CEC). By leveraging its learned representations and predictive capabilities, we not only enhance traditional QoR metrics but also introduce a novel paradigm of orientational optimization.

TABLE VI
AREA-DELAY COMPARISON, TOTAL RUNTIME COMPARISON IN LOGIC OPTIMIZATION BETWEEN DRILLS AND ITS MODIFICATION.

Benchmark	$Resyn2'$		Original DRiLLS				CircuitFlow + DRiLLS					
	Area (um^2)	Delay (ps)	Area (um^2)	Delay (ps)	Time ($min.$)	Best iteration	Area (um^2)	Delay (ps)	Time ($min.$)	Best iteration	ADP Impr.	Time Impr.
arbiter	7495.29	1036.38	7495.29	1036.38	48.66	3	7495.29	1036.38	9.70	1	0%	**5.02**×
priority	671.61	2361.86	548.67	1661.08	32.71	48	523.48	1409.22	8.07	7	23.54%	**4.05**×
cavlc	463.76	280.75	464.23	274.42	9.71	23	463.53	274.42	5.72	3	0.15%	**1.70**×
int2float	147.67	151.72	148.13	139.94	9.56	10	146.5	140.47	8.97	4	0.73%	**1.06**×
voter	15519.19	968.11	14315.13	889.31	33.23	62	14179.22	896.55	10.90	5	0.14%	**3.04**×
mem_ctrl	31716.05	3598.3	31686.19	3543.12	86.97	76	31749.64	3534.47	17.36	6	0.04%	**5.01**×
dec	290.2	112.91	290.2	112.91	9.19	5	290.2	112.91	6.96	1	0%	**1.32**×
i2c	900.23	276.08	928.69	235.16	9.92	15	902.33	238.05	5.85	2	1.67%	**1.70**×
router	161.9	245.18	135.54	214.26	8.95	20	143	187.76	7.32	8	8.16%	**1.22**×
ctrl	83.98	160.88	83.98	136.51	32.70	74	83.75	102.28	9.41	40	33.83%	**3.47**×
Avg.	5744.988	919.217	5609.605	824.309	28.16	33.6	5597.694	793.251	9.026	7.7	3.97%	**3.11**×

TABLE VII
QOR AND RUNTIME COMPARISON BETWEEN CBTUNE AND ITS MODIFICATION.

benchmark	CBtune			CircuitFlow + CBtune			
	Area (um^2)	Delay (ps)	Time (s)	Area (um^2)	Delay (ps)	Time (s)	ADP Impr.
arbiter	7495.29	1036.38	27.82	7495.29	1036.38	36.31	0.00%
priority	598.22	1850.53	21.21	558.6	1354.89	26.56	46.27%
cavlc	459.1	284.51	21.95	463.53	274.42	32.07	2.69%
int2float	147.67	151.72	18.65	147.67	143.63	26.01	5.63%
voter	14154.5	948.25	28.12	14182.49	946.93	37.57	-0.06%
mem_ctrl	31546.92	3752.8	30.72	31610.43	3694.3	39.73	1.38%
dec	290.2	112.91	21.97	290.2	112.91	31.39	0.00%
i2c	896.96	269.56	20.77	896.96	267.98	31.1	0.59%
router	145.05	254.84	18.13	136.24	216.57	25.9	25.28%
ctrl	83.98	122.63	20.27	83.98	104.32	29.3	17.55%
Avg.	5581.79	878.41	22.96	5586.539	815.23	31.59	7.11%

A. Optimization Sequence Generation

In logic synthesis, the optimization process aims to enhance the QoR of the mapped netlist. The efficacy of this process is highly dependent on the sequence in which different optimization algorithms are applied. To address this challenge, we leverage the predictive capabilities of our model by integrating it separately into DRiLLS [13], a reinforcement learning framework, and CBtune [14], a bandit-based approach, to generate optimal optimization sequences.

1) Experiment Settings: Our integration involves three key modifications to the original DRiLLS framework:

- State Representation: We replace the original AIG state representation in DRiLLS and CBtune with the graph embeddings generated by CircuitFlow. These embeddings are then fed into the Actor network (in DRiLLS) or bandit agent (in CBtune) to inform the policy for selecting the next action (i.e., the next optimization algorithm).
- QoR Prediction: During each iteration of the sequence generation, we use CircuitFlow to rapidly predict the QoR of the circuit after applying a candidate optimization. This avoids the time-consuming process of running the actual synthesis tool at each step. For final validation, the QoR of the complete 10-step sequence is computed using the standard ABC command to ensure accuracy.
- Action Space: To maintain consistency with the training domain of CircuitFlow, the set of available optimization commands is limited to $\{rewrite, refactor, balance\}$. The sequence length is fixed at 10 to provide a fair comparison against the revised script $resyn2'$.

To evaluate the efficacy of our method, we conducted experiments on the `EPFL benchmarks` [32] using the `asap7.lib` technology library. We set the maximum iterations in DRiLLS to 100 for full exploration and set the delay constraint to a sufficiently large value to relax timing requirements, while keeping other experimental settings consistent with the original implementations of DRiLLS [13] and CBtune [14].

2) Experiment Result: **Analysis of DRiLLS integration :** Table VI compares our CircuitFlow-integrated DRiLLS against the baseline $resyn2'$ and the original DRiLLS. The analysis reveals two major findings: (1) CircuitFlow DRiLLS achieves a notable average improvement of 3.97% in Area-Delay Product (ADP) over the original DRiLLS. This demonstrates that the graph embeddings from CircuitFlow are effective circuit representations for the reinforcement learning agent. For certain circuits, the improvement is substantial, such as `ctrl` (33.83%) and `priority` (23.54%); and (2) More significantly, CircuitFlow + DRiLLS shows a reduction both in runtime and iteration. The primary reason for this speedup is that the fast and accurate QoR prediction from CircuitFlow enables the RL agent to find a high-quality solution in fewer iterations, as shown in the "Best iteration" in Table VI. Besides, avoiding the time-consuming process of running the actual synthesis tool at each step also reduces the total consuming time. For example, for `cavlc`, the original DRiLLS took 48 iterations, whereas the CircuitFlow + DRiLLS finds a better solution in just 7 iterations. This efficiency not only validates the predictive accuracy of CircuitFlow but also proves the practical value of representative learning models in downstream tasks.

Analysis of CBtune integration: Table VII shows the comparison result of CBtune and highlights the impact of the enhanced contextual information. While in DRiLLS experiment, where runtime reduction was a key benefit, integrating CircuitFlow introduces a slight time overhead due to model inference and executing at each search step. However, this trade-off proves highly beneficial for the quality of the final result. For instance, CircuitFlow + CBtune achieves a remarkable 46.27% ADP improvement for the `priority` benchmark

and 25.28% for `router`. This experiment demonstrates that CircuitFlow can serve not just as a state representation for deep RL, but also as a contextual information provider for other learning algorithms, empowering them to find higher-quality solutions.

B. Orientational Optimization

Different from existing methods that merely predict the best achievable Quality of Results (QoR), CircuitFlow enables a novel paradigm: *Orientational Optimization*. This task aims to steer a circuit's evolution toward a specific target structural pattern. Formally, given a starting circuit G_{start} and a target circuit G_{target} (or a target structural embedding h_{target}), the objective is to find an optimization sequence $\mathcal{R} = (a_1, a_2, \cdots, a_n)$ such that the structural distance between the transition circuit G_{trans} and G_{target} is minimized.

1) Methodology and Lookahead Mechanism: To solve this exploration problem, we integrate CircuitFlow into the Advantage Actor-Critic (A2C) framework. The detailed workflow is as follows:

1) **Target Definition:** We first compute the motif-based embedding of the start and target circuit, denoted as M_{start} and M_{target}.
2) **Action Selection via Prediction:** We utilize the pre-trained DAGTrans to encode the current circuit G_{start} into a high-dimensional embedding. This embedding serves as the state observation for the RL agent:
$$s_t = h_t = T_{DAG}(G_{start}) \tag{14}$$
3) **A2C Decision:** We employ an Actor-Critic agent to navigate the optimization space. At each timestep t, the Actor network $\pi(a|s;\theta)$ takes the current state embedding s_t as input and samples an optimization action a_t from the learned policy distribution:
$$a_t \sim \pi(a_t|s_t;\theta) \tag{15}$$
4) **CircuitFlow-Guided Reward Mechanism:** To guide the agent toward the target structure, we design a dense reward based on the CircuitFlow motif embeddings. We calculate the cosine similarity improvement between the current motif distribution M_t and the target M_{target}:
$$\begin{aligned} M_t &= \text{MLP}_{motif}(T_{OPT}(h_t, a_t)) \\ r_t &= \left(1 - \frac{\text{Cos}(M_t, M_{target})}{\text{Cos}(M_{start}, M_{target})}\right) - g_{t-1} \end{aligned} \tag{16}$$
where g_{t-1} is the similarity score of the previous step. This reward signal incentivizes the agent to choose actions that maximize structural alignment with the target.

2) Experiment Settings: We implement the proposed framework based on the open-source logic synthesis reinforcement learning environment, DRiLLS [13]. To ensure a fair comparison and robust evaluation, the experimental setup is configured as follows:

- **Implementation Details:** The Actor and Critic networks are implemented as Multi-Layer Perceptrons (MLPs) with two hidden layers. The input dimension matches the CircuitFlow embedding size.

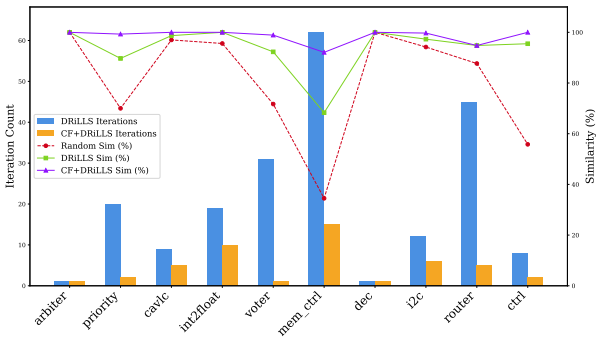


Fig. 9. Performance comparison on similarity and iteration of orientational optimization.

- **Episode Configuration:** To impose a constraint on the optimization cost, we set the maximum episode length to $T = 10$ steps.
- **Target Generation:** The target circuits (G_{target}) are generated by applying random optimization sequences (length=10) to circuits selected from the EPFL benchmark suite. This ensures the targets are reachable but structurally distinct from the start states.
- **Episode Termination Criteria:** To balance exploration efficiency and computational cost, an episode terminates if either of the following conditions is met:
  1) **Convergence:** The cosine similarity between the current motif distribution M_t and the target M_{target} exceeds **95%**, indicating successful structural alignment.
  2) **Time Limit:** The agent reaches the maximum step limit of $T = 10$.
- **Baselines:** We compare our CircuitFlow-guided A2C agent against two baselines: (1) **Random Exploration**, which samples valid optimization recipes uniformly; and (2) **Standard DRiLLS**, which utilizes standard A2C network.

3) Experiment Result: Fig. 9 compares the performance of iteration and similarity across ten benchmarks. The baseline random strategy is to generate the recipe randomly and achieves only moderate similarity in most cases, e.g., 70% for `priority`, 71.7% for `voter`, and 55.8% for `ctrl`. Additionally, the result of random generation is the best of 50 iterations, and in some iterations, the similarity can be as low as 20% for some circuits. In contrast, the modified DRiLLs method significantly improves the similarity in some iterations. With most of them are reaching a similarity over 90%, it proves the feasibility and availability of orientational optimization. For instance, for `priority` and `ctrl`, similarity increase from 70% to 89.7% and from 55.8% to 95.5% after several iterations of RL agent, respectively. Furthermore, the CircuitFlow + DRiLLS, which can predict the motif number of next optimization, consistently achieve the highest similarity, reaching nearly 100% similarity nearly all benchmarks. For `priority`, `cavlc`, `int2float` and `ctrl`, the same graph (100% similarity) is obtained. Importantly, the improvement is achieved within minimal iterations, showing that the motif-guided optimization effectively accelerate the optimization process towards the required circuits.

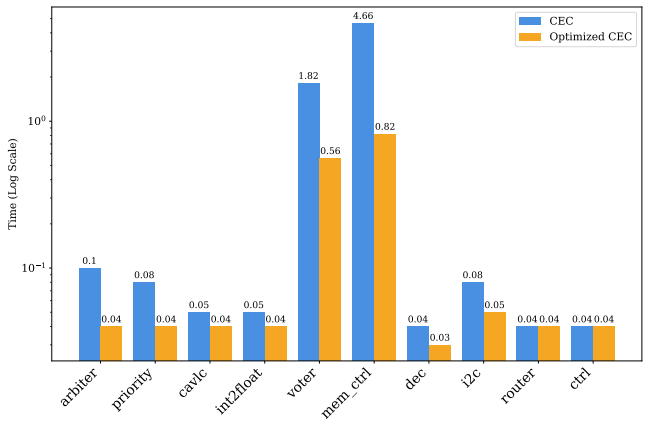


Fig. 10. CEC time comparison.

In summary, the RL framework significantly outperforms random exploration in orientational optimization, while the CircuitFlow enhanced motif prediction can further accelerate convergence and steers optimization towards the desired AIG structures with near-perfect similarity.

4) Application: Beyond static prediction, our framework offers a novel approach to enhancing the CEC process. CEC traditionally verifies if two circuits are functionally identical, often relying on computationally intensive methods like simulation, SAT sweeping [33], or structural hashing. A key insight is that structural similarity between circuits significantly eases CEC by structural hashing [33], [34]. Therefore, by making circuits more structurally alike, we can accelerate the equivalence verification. We leverage this by integrating our framework into a similarity-driven optimization pipeline. When two circuits require equivalence checking, we first compute their structural similarity using our learned embeddings. If this similarity is below a certain threshold, we apply orientational optimization to the larger circuit, guiding its transformation towards smaller circuit. To evaluate this, we conducted experiments in two steps:

- Baseline CEC: We first measured the CEC time for a set of benchmark circuits against optimized target circuits generated by random optimization recipes.
- Optimization CEC: Subsequently, we applied our orientational optimization to steer the benchmarks towards these same target circuits and then re-measured the CEC time between the newly optimized benchmarks and their respective targets.

Fig. 10 shows the runtime of standard CEC and our optimized CEC framework. Overall, our method demonstrates a clear improvement in equivalence checking efficiency on most benchmarks. In most cases, the runtime is largely reduced at ranging from 20% to 82%, such as `cavlc` and `arbiter`. But in `router` and `voter`, there is no improvement, probably because the scale of the circuit is limited. On the other hand, `mem_ctrl` and `voter`, which are larger than other circuits, have greater improvements, indicating that orientational optimization may be more effective when the circuit is larger.

In summary, these results confirm that increasing structural similarity through our orientational optimization can significantly reduce the runtime of equivalence checking in most cases, thereby validating the practical benefit of integrating our framework into CEC.

VIII. Conclusions and Discusions

In this work, we propose **CircuitFlow** to bridge the semantic gap in dynamic logic optimization. By integrating motif-aware static encoding (**DAGTrans**) with dynamic latent simulation (**OPTrans**), the framework effectively captures circuit evolution, reducing physical metric prediction loss by up to 72% and accelerating downstream optimization by 3.11×. Future work envisions three key directions: (1) exploring scalable representation methodologies to efficiently extend CircuitFlow to ultra-large, industrial-scale circuits; (2) incorporating a broader spectrum of optimization operators and extended recipe horizons while mitigating cumulative predictive errors; and (3) exploring additional application scenarios for CircuitFlow within the logic synthesis flow.

References

[1] Z. Shi, H. Pan *et al.*, "DeepGate2: Functionality-Aware Circuit Representation Learning," in *Proc. of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[2] Z. Shi, Z. Zheng *et al.*, "DeepGate3: Towards Scalable Circuit Representation Learning," *CoRR*, vol. abs/2407.11095, 2024.

[3] Z. Wang, C. Bai *et al.*, "Functionality matters in netlist representation learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022, p. 61–66.

[4] S. Khan, Z. Shi *et al.*, "DeepSeq: Deep Sequential Circuit Learning," in *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2024, pp. 1–2.

[5] ——, "DeepSeq2: Enhanced Sequential Circuit Learning with Disentangled Representations," in *Proc. of IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025, p. 498–504.

[6] R. Karimi, F. Faez *et al.*, "Logic synthesis optimization with predictive self-supervision via causal transformers," 2025. [Online]. Available: https://arxiv.org/abs/2409.10653

[7] M. Zhao, J. Liu *et al.*, "Mils: Modality interaction driven learning for logic synthesis," in *Proceedings of the Great Lakes Symposium on VLSI 2025*, ser. GLSVLSI '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 64–70. [Online]. Available: https://doi.org/10.1145/3716368.3735203

[8] A. B. Chowdhury, B. Tan *et al.*, "OpenABC-D: A Large-Scale Dataset For Machine Learning Guided Integrated Circuit Synthesis," *CoRR*, vol. abs/2110.11292, 2021.

[9] K. Zhu, M. Liu *et al.*, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2020, p. 145–150.

[10] Y. Cai, R. Wang *et al.*, "AiLO: A Predictive Framework for Logic Optimization Using Multi-Scale Cross-Attention Transformer," *ACM Transactions on Design Automation Electronic Systems*, 2025.

[11] Z. Wang, C. Bai *et al.*, "FGNN2: A Powerful Pretraining Framework for Learning the Logic Functionality of Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 1, pp. 227–240, 2025.

[12] J. Liu, J. Zhai *et al.*, *PolarGate: Breaking the Functionality Representation Bottleneck of And-Inverter Graph Neural Network*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3676536.3676834

[13] A. Hosny, S. Hashemi *et al.*, "DRiLLS: Deep Reinforcement Learning for Logic Synthesis," in *Proc. of IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 581–586.

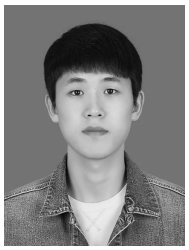
[14] F. Liu, Z. Pei *et al.*, "Cbtune: Contextual bandit tuning for logic synthesis," in *Proc. of IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[15] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. of IEEE/ACM Design Automation Conference (DAC)*, 2006.

[16] A. Mishchenko, R. K. Brayton *et al.*, "Delay optimization using SOP balancing," in *Proc. of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2011.
[17] C. Deng, Z. Yue *et al.*, "Less is More: Hop-Wise Graph Attention for Scalable and Generalizable Learning on Circuits," in *Proc. of IEEE/ACM Design Automation Conference (DAC)*. ACM, 2024, pp. 175:1–175:6.
[18] R. Ma, Y. Zhou *et al.*, "Dynamicrtl: Rtl representation learning for dynamic circuit behavior," 2025. [Online]. Available: https://arxiv.org/abs/2511.09593
[19] Z. Zheng, S. Huang *et al.*, "Deepgate4: Efficient and effective representation learning for circuit design at scale," 2025.
[20] J. K. Matelsky, E. P. Reilly *et al.*, "DotMotif: an open-source tool for connectome subgraph isomorphism search and graph queries," *Scientific Reports*, vol. 11, no. 1, Jun 2021.
[21] H. Gao and S. Ji, "Graph U-Nets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 9, pp. 4948–4960, 2022.
[22] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 1024–1034.
[23] Y. Luo, V. Thost, and L. Shi, "Transformers over directed acyclic graphs," *Advances in Neural Information Processing Systems*, vol. 36, pp. 47 764–47 782, 2023.
[24] J. Yang, Z. Liu *et al.*, "GraphFormers: GNN-nested Transformers for Representation Learning on Textual Graph," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
[25] M. Li, S. Khan *et al.*, "DeepGate: learning neural representations of logic gates," in *Proc. of IEEE/ACM Design Automation Conference (DAC)*. ACM, 2022, pp. 667–672.
[26] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Proceedings of Computer Aided Verification*. Springer Berlin Heidelberg, 2010, pp. 24–40.
[27] L. T. Clark, V. Vashishtha *et al.*, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.
[28] N. Wu, Y. Li *et al.*, "Gamora: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks," in *Proc. of IEEE/ACM Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.
[29] J. Yang, Z. Liu *et al.*, "Graphformers: Gnn-nested transformers for representation learning on textual graph," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 798–28 810, 2021.
[30] D. P. Kingma and J. Ba, "ADAM: A Method for Stochastic Optimization," in *Proc. of International Conference on Learning Representations (ICLR)*, 2015.
[31] V. Thost and J. Chen, "Directed Acyclic Graph Neural Networks," in *Proc. of International Conference on Learning Representations (ICLR)*. OpenReview.net, 2021.
[32] L. Amarù, P. E. Gaillardon *et al.*, "The EPFL Combinational Benchmark Suite," in *International Workshop on Logic Synthesis*, 2015.
[33] A. Mishchenko, S. Chatterjee *et al.*, "Improvements to combinational equivalence checking," in *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006, p. 836–843.
[34] A. Kuehlmann, V. Paruthi *et al.*, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

Miao Liu received the B.S. degree in Data Science from Peking University, Beijing, China, in 2020. He is currently pursuing the Ph.D. degree with the University of Chinese Academy of Sciences, Beijing, China. His current research interests include logic optimization, reinforcement learning, and AI for EDA.

Xinhua Lai received the M.S. degree in Computer Science from Central South University, Changsha, China, in 2019. He is currently pursuing the Ph.D. degree with the University of Chinese Academy of Sciences. His research interests include design space exploration, machine learning, deep learning, large language model and optimization methods with applications in VLSI.

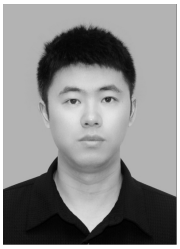


Liwei Ni is currently pursuing the Ph.D. degree with the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, and also with Pengcheng Laboratory. His current research interests include logic optimization, technology mapping, and formal verification.

Xingyu Meng received the PhD degree from the department of Electrical and Computer Engineering at the University of Texas at Dallas as part of the Trustworthy and Intelligent Embedded System (TIES) lab. His research interests include hardware and system security, Trojan detection, and hardware verification. His research has been published in Design Automation Conference (DAC), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), etc.

Rui Wang received the B.S. degree in Mathematics and Applied Mathematics from Tianjin Normal University, Tianjin, China, in 2023. He is currently pursuing the M.S. degree in Computer Science and Technology at the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China. His research interests are focused on logic synthesis.

Junfeng Liu received the PhD degree in computer science from Beihang University, China, in 2024. He is currently a postdoctoral researcher at Pengcheng Laboratory, China. His research interests include EDA logic synthesis and graph data management. He has published over 10 papers in journals and conferences such as TCAD, TKDE, TODAES, ICCD, WSDM, and CIKM.



Xingquan Li (Member, IEEE) received the B.E. and Ph.D. degree from Fuzhou University, Fuzhou, China, in 2013 and 2018, respectively. He is an Associate Professor at Pengcheng Laboratory. His research interests include EDA and AI for EDA. His team has developed an open-source iEDA infrastructure/toolchain. He has published over 70 papers, and received three First Place Awards from ICCAD@CAD Contest in 2017, 2018, and 2022. In 2020, he received the Application Award of Operations Research from the Operations Research Society of China, and the Best Paper Award from ISEDA 2023.

Jungang Xu (Member, IEEE) received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2003. He is a Professor at the University of Chinese Academy of Sciences. His research interests include multimodal intelligence, AI for EDA, and embodied AI. He is the Director of the Cloud Computing & Intelligent Information Processing Lab.