GraphMT – QoR(delay)prediction of logic synthesis of an AIG using graph representation and masked transformers

Soumajit Roy
244101057
r.soumajit@iitg.ac.in

Shashank Aggarwal
244101050
ashashank@iitg.ac.in

Shwetank Pratap Tiwari
244101055
t.shwetank@iitg.ac.in

Shivaram M
244101053
m.shivaram@iitg.ac.in

Abstract

Logic synthesis optimization sequences are crucial for meeting circuit area and delay specifications but are time-consuming to execute. We have used a deep learning method to predict the Quality of Results (QoR) for unseen circuit-optimization sequence pairs. Our approach translates structure transformations into vectors using embedding methods and leverages Transformer architecture to extract features from optimization sequences. To enable generalization across different circuits, we represent circuit graphs as adjacency and feature matrices, using Graph Neural Networks (GNNs) to extract structural features. Through benchmarking various combinations of Transformer and three typical GNNs in a joint learning framework, our experimental results demonstrate that the integration of Transformer with GraphSage yields the most effective performance for QoR prediction.

Despite these advancements, existing models face challenges including overfitting due to limited public circuit availability and expressiveness limitations of graph encoders. To address these issues, we introduce GraphMT, an approach that harnesses autoregressive transformer models and predictive self-supervised learning (SSL) to forecast QoR trajectories. GraphMT integrates cross-attention modules to effectively merge insights from circuit graphs and optimization sequences, enhancing prediction accuracy for QoR metrics. Additionally, we improve performance on unseen And-Inverter Graphs (AIGs) by implementing a fine-tuning approach inspired by ControlNet, which employs two model copies—one frozen to maintain the initial knowledge base during fine-tuning—thereby preserving generalization capabilities while adapting to new circuit designs.

The code is available in our GitHub Repository.

Keywords Deep learning · Graph Neural Networks · Logic Synthesis Optimization

1 Introduction

Logic synthesis plays a critical role in the electronic design automation (EDA) flow, acting as an intermediate layer that optimizes a given Boolean network and maps it to a gate-level netlist while aiming to enhance the overall quality-of-results (QoR). In the pre-mapping phase of logic synthesis, various structural transformations are applied to improve circuit performance, area, and delay. These transformations are typically implemented on circuits represented as And-Inverter Graphs (AIGs), as adopted by the state-of-the-art academic logic synthesis tool ABC. To achieve optimal QoR, engineers often combine multiple structural transformations into logic synthesis optimization sequences. However, selecting an effective optimization sequence remains a time-consuming and heuristic-driven process, largely relying on expert knowledge and extensive iterative testing.

One of the key challenges in logic synthesis lies in the lack of predictability in evaluating the performance of different optimization sequences. This unpredictability creates a need for methods that can estimate the QoR of a given circuit-optimization sequence pair without requiring full synthesis runs. Addressing this challenge, recent advancements in deep learning (DL) offer promising solutions by learning complex input-output mappings and enabling accurate predictions without explicit mathematical models.

Here we have used a deep learning-based approach to predict the QoR of unseen circuit-optimization sequence pairs, thereby significantly accelerating the logic synthesis process. Our method leverages advanced natural language processing (NLP) techniques, specifically the Transformer architecture, to model and extract meaningful features from optimization sequences, which are represented as embedded vectors. Simultaneously, we represent circuit structures using an adjacency matrix and feature matrix, and employ Graph Neural Networks (GNNs) to capture their topological and structural characteristics.

To ensure generalization across different circuits and optimization sequences, we explore a **joint learning framework** that combines the Transformer with three representative GNN architectures. Our experimental results demonstrate that the combination of **Transformer and GraphSAGE** yields the best performance in QoR prediction, as measured by Mean Absolute Error (MAE). This joint learning approach provides a scalable and accurate solution for predicting synthesis outcomes, thus enabling faster and more informed decision-making in the logic synthesis workflow.

The Highlights of our joint learning approach are summarized below:

- An optimization sequence feature extractor based on the Transformer NLP model. The input is a tokenized optimization sequence transformed via embedding, and the output is a feature vector suitable for downstream QoR prediction.
- A circuit feature extractor is introduced using three typical GNN architectures (GCN, GAT, and GraphSAGE), extracting circuit structure features from the adjacency and feature matrices.
- A joint learning policy is developed that integrates both the optimization sequence and circuit structure features to predict the performance of previously unseen circuit-optimization sequence pairs.
- Comprehensive experiments are conducted across nine model combinations, with insights and recommendations provided for different logic synthesis scenarios to guide future research.

Previous methods suffer from a key limitation in the recipe encoder, which processes data independently of the AIG (And-Inverter Graph) representations. This isolation introduces an information bottleneck, especially noticeable when conventional concatenation is used to merge recipe and AIG embeddings. To overcome this, we introduce a decoder-only transformer model that leverages a joint loss function during training. This design enables the model to contextualize heuristic embeddings by attending to AIG representations through a Cross-Attention mechanism, leading to a more cohesive integration of information. Our contributions to address these challenges are as follows:

- **Hierarchical Graph Pooling**: A graph pooling strategy optimized for DAG (Directed Acyclic Graph) structures, converting AIGs into sequential embeddings efficiently.

- **QoR Progress Estimation**: An auxiliary self-supervised task that predicts intermediate QoR (Quality of Results) values during training, guiding the model's learning process.
- **Causal Transformer Architecture**: A decoder-only transformer that sequentially predicts QoR outcomes by interpreting both recipe steps and AIG representations.
- **Cross-Attentive Fusion**: A fusion mechanism using cross-attention to effectively align and contextualize recipe and AIG embeddings, allowing the model to dynamically identify relevant graph regions per heuristic.

ControlNet maintains the performance and integrity of a large pretrained model by freezing its parameters and introducing a trainable replica of its encoder layers. This setup leverages the original model as a robust backbone for learning various conditional control signals. The architecture connects the frozen and trainable components through zero-initialized convolution layers. These layers start with zero weights and gradually adapt during training, allowing the network to learn without immediately disrupting the deep feature representations. This design helps prevent the introduction of harmful noise at early stages and safeguards the pretrained backbone from degradation during fine-tuning.

2 Background Work

Ground Truth Generation: In our EDA workflow, we optimize logic-level circuits represented by AIG graphs G using a sequence of AIG operations, referred to as a *recipe*, through time-consuming tools like ABC. We use *levels* (representing delay) as the QoR metric. The final QoR value, denoted as y, is obtained after applying the full recipe and serves as the ground truth. Our model aims to predict this value y given the input AIG and the recipe during inference.

2.1 Final QoR Prediction

The goal of the QoR prediction task is to determine a learnable function f_γ that takes a pair consisting of a recipe and an AIG graph, (G, r_i), and outputs a numerical value $\hat{y}$. This value corresponds to the predicted final ground truth QoR, which, in our case, is the delay. The task can be represented as:

$$\hat{y} = f_\gamma(G, r_i) \tag{1}$$

The loss function, which uses Mean Squared Error (MSE), is calculated as:

$$\text{Loss} = \sum_{k=1}^{M} \text{MSE}(\hat{y}_k, y_k) \tag{2}$$

Here, y_k denotes the ground truth QoR for the k-th step.

2.2 GNN Basics

Graph Neural Networks (GNNs) [30] are specialized neural models designed to handle Graph structured data. When learning node representations, each node v in the graph is associated with an initial feature vector $\mathbf{x}_v$. The GNN learns a corresponding d-dimensional embedding $\mathbf{h}_v$, which encodes information not only from the node itself but also from its surrounding neighbors in the graph.

$$\mathbf{h}_v = f\left(\mathbf{x}_v, \mathbf{x}_{cc}[v], \mathbf{h}_{nb}[v], \mathbf{x}_{nb}[v]\right) \tag{3}$$

Here,

- $\mathbf{x}_{cc}[v]$ denotes the features of the edges connected to node v,

- $\mathbf{h}_{nb}[v]$ is the set of embedding features of the neighboring nodes of v,
- $\mathbf{x}_{nb}[v]$ is the set of input features of the neighboring nodes of v,
- f is the transfer function that projects these inputs into the d-dimensional space.

Since the goal is to compute a unique solution for $\mathbf{h}_v$, the Banach Fixed Point Theorem can be employed. This enables the equation to be reformulated as an iterative update procedure for determining the node embeddings.

$$\mathbf{H}^{(t+1)} = \mathcal{F}(\mathbf{H}^{(t)}, \mathbf{X}) \tag{4}$$

where $\mathbf{H}$ and $\mathbf{X}$ denote all the hidden states $\mathbf{h}$ and input features $\mathbf{x}$, respectively.

The output of the GNN is then calculated by passing the state $\mathbf{h}_v$ as well as the feature $\mathbf{x}_v$ to an output function g:

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v) \tag{5}$$

Both f and g can be interpreted as feedforward fully connected (FC) neural networks.

Graph Neural Networks (GNNs) perform various tasks based on different components of a graph, such as nodes, edges, subgraphs, and the entire graph. In this work, GNNs are employed to learn circuit graph embeddings that capture the structural properties of the circuits. These embeddings are then used for predicting the Quality of Results (QoR) in downstream tasks.

2.3 CNN Basics

A convolutional neural network (CNN) [31] consists of a feature extraction component, which typically includes a convolutional layer followed by a subsampling (or pooling) layer. The convolutional layer is composed of multiple feature maps, each of which is made up of neurons arranged in a grid-like structure. Each neuron is only connected to a subset of nearby neurons, not the entire set. In a given feature map, the neurons share the same set of weights, which correspond to the convolutional kernel. This kernel is typically initialized with random values and is updated during the network's training process to learn optimal weights.

One of the key advantages of weight sharing (via the convolutional kernels) is that it significantly reduces the number of parameters in the network, which in turn helps to mitigate the risk of overfitting. The subsampling layer, which may use methods like mean pooling or max pooling, further reduces the size of the data representation. Subsampling can be thought of as a specialized form of convolution, and both convolution and subsampling play crucial roles in simplifying the model and lowering its parameter count, thereby enhancing efficiency.

2.4 LSTM Basics

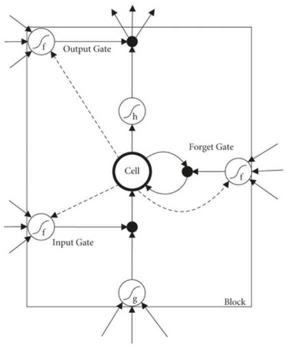


Figure 1: LSTM unit illustration.

Long Short-Term Memory (LSTM) [32] is a type of recurrent neural network (RNN) designed to address the problem of learning long-term dependencies. It achieves this by incorporating control gates within its recurrent units. As depicted in Figure 1, an LSTM unit generally includes a cell, an input gate, an output gate, and a forget gate. The input gate takes in data from sequences and other parts of the network, and it is trained to decide when to add information to the cell. The output gate, like the input gate, learns when to release the information stored in the cell. The forget gate, on the other hand, determines when the information in the cell is no longer necessary and should be discarded. These gates use a sigmoid activation function and pointwise multiplication to selectively allow or prevent the flow of information.

2.5 Transformer Basics

The Transformer network architecture, introduced by Ashish Vaswani et al., was originally designed for machine translation tasks [27]. Unlike previous models such as RNNs or CNNs, both the encoder and decoder in the Transformer architecture are built entirely on attention mechanisms. This design enables the model to focus on different positions in the input sequence while processing each input vector, using self-attention to capture relevant relationships for improved encoding.

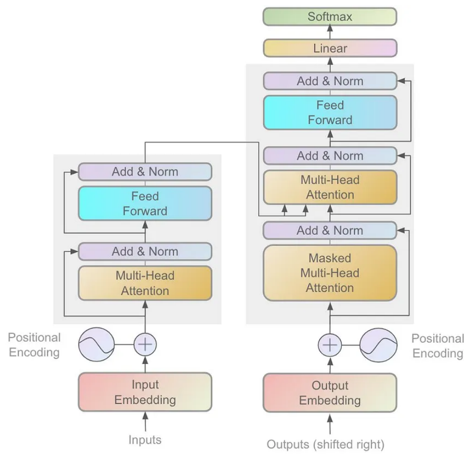


Figure 2: Transformer Architecture.

The basic structure of the Transformer is illustrated in Figure 2, with the left half representing the Encoder and the right half representing the Decoder. The Transformer consists of six layers of this architecture. In each layer, the Encoder converts the input sequence into an encoded representation, and the Decoder generates a new sequence as output based on this encoded representation.

However, since this work does not involve converting an input sequence into another sequence, the standard application of the Transformer model is not suitable. Instead, Section 4.1.2 demonstrates how partial components of the Transformer can be utilized to extract features from the optimization sequence.

3 Motivation

Structural transformations applied during circuit optimization often exhibit interdependencies—when used together, they can yield varying optimization outcomes. For instance, in the ABC framework, certain transformations like zero-cost rewrites (`rewrite -z`) do not directly reduce the number of nodes in the AIG (And-Inverter Graph), but rely on subsequent transformations to achieve full optimization. The self-attention mechanism in Transformers, which computes relationships between all elements in a sequence, is inherently well-suited for capturing such dependencies in optimization sequences.

Moreover, since circuit representations like AIGs are inherently graph-based and many transformation algorithms operate on these structures, a graph-centric approach becomes essential. While earlier methods have focused solely on the optimization sequence, neglecting the circuit's structural information, this work addresses that gap. We present a hybrid approach that combines a Graph Neural Network (GNN) with a joint learning strategy to effectively capture the structural features of circuits along with the sequential dynamics of optimization transformations. The objective is to accurately predict the Quality of Results (QoR) for circuit-optimization sequences that the model has not encountered before.

4 Methodology

The input and the architecture of the model for neural networks are mainly described. The input is the optimization sequence, which is converted into a vector representation after the embedding layer. The neural network architecture consists of an optimization sequence feature extractor and a circuit feature extractor which are used to predict the QoR of the optimization sequence for different circuits after adopting a joint learning policy.

4.1 Recipe Sequence Feature Extraction

The implementation of an Recipe sequence feature extractor based on the Transformer is presented here. In addition, for a comprehensive comparison, the classical LSTM and the CNN are implemented as well.

4.1.1 Sequence Embedding

In our project, the input optimization sequence consists of say, seven ABC structure transformations: refactor, refactor-z, rewrite, rewrite-z, resub, resub-z, and balance. The sequence is assumed to be [rewrite, resub, refactor, balance]. Here's how we convert this sequence into a format that can be processed by a neural network.

First, we treat each of the seven structural transformations as unique entities, similar to words in a natural language context. Each transformation is assigned an index between 0 and 6. To improve the neural network's learning process, we create an embedding matrix where each index is mapped to a vector of three latent factors, meaning each transformation is represented by a vector of length 3.

For illustration purposes, we randomly initialize the embedding matrix for the seven structural transformations. This allows the model to learn from the relationships between different transformations and their impact on the optimization process.

$$\begin{bmatrix} -0.4093 & -1.1011 & 0.0790 \\ -0.2704 & 0.0708 & 0.6557 \\ -0.5706 & -0.2703 & 2.2453 \\ 0.6731 & -0.6557 & -0.9846 \\ -1.1936 & -0.0705 & 0.3704 \\ 0.2741 & 0.4531 & 2.5046 \\ 0.4327 & -0.9486 & 2.2643 \end{bmatrix}$$

The row numbers of the matrix correspond to the index values, and each row vector represents the embedding vector corresponding to the index. As an example, an optimization sequence [rewrite, resub, refactor, balance] can be considered as a sentence composed of a dictionary. By querying the indexes in the dictionary, the optimization sequence can be transformed to [2, 4, 0, 6]. By querying rows 2, 4, 0, 6 of the embedding matrix by indexes, respectively, the optimization sequence is represented as the following embedding vector:

$$\begin{bmatrix} -0.5706 & -0.2703 & 2.2453 \\ -1.1936 & -0.0705 & 0.3704 \\ -0.4093 & -1.1011 & 0.0790 \\ 0.4327 & -0.9486 & 2.2643 \end{bmatrix}$$

Each row represents a structural transformation performed at a one-time step. Note that not every structural transformation is replaced by a vector, but is replaced by the index used to find the vectors in the embedding matrix. The values of the entire embedding matrix are used as part of the training. As the model is trained, the embedding vector gets the appropriate values.

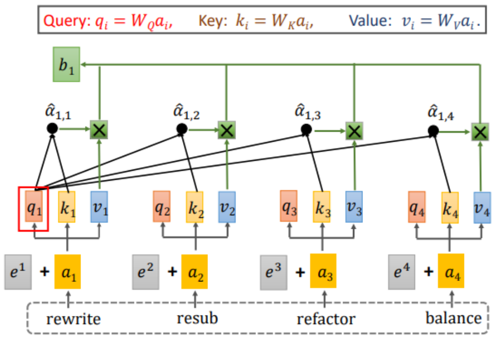


Figure 3: optimization sequence feature extractor based on self-attention.

4.1.2 Transformer Based Recipe Feature Extraction

Here, how to use the partial modules in the transformer for optimization sequence feature extraction will be presented.

a. Computation of self-attention
As the core component of the transformer, we first explain how the self-attention mechanism extracts optimization sequence features. Figure 3 illustrates how structural transformations in the sequence [rewrite, resub, refactor, balance] are converted into vectors $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, and $\mathbf{a}_4$ through optimization sequence embedding (described in Section 4.1.1), with position information e^i added to each vector.

For position encoding, we adopt the approach from [16], which simplifies the original transformer method. Similar to word vector generation, position encoding is initialized and then refined during training to create a unique position vector for each position.

The self-attention computation follows several steps. Initially, each input vector $\mathbf{a}_i$ is transformed by parameter matrices $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ to produce query vector $\mathbf{q}_i$, key vector $\mathbf{k}_i$, and value vector $\mathbf{v}_i$.

Next, scores are calculated. Figure 3 shows how the self-attention vector is computed for "rewrite" (the first transformation). During this process, every transformation in the input sequence contributes to scoring "rewrite." These scores determine the attention distribution across the sequence when encoding the "rewrite" transformation.

The score is calculated as the dot product between the query vector of "rewrite" and the key vector of each transformation. For example, the first score comes from the dot product of $\mathbf{q}_1$ and $\mathbf{k}_1$, while the second score is derived from $\mathbf{q}_1$ and $\mathbf{k}_2$. To ensure smoother gradients, we divide the score by $\sqrt{d}$, where d represents the dimension of the query and key vectors:

$$\alpha_{1,i} = \frac{\mathbf{q}_1 \cdot \mathbf{k}_i}{\sqrt{d}} \tag{4}$$

The scores are then normalized using the softmax function to obtain softmax scores:

$$\bar{\alpha}1, i = \frac{\exp(a1, i)}{\sum_j \exp(a_{1,j})} \tag{5}$$

This normalization ensures all scores are positive and sum to 1. In the final step, each value vector $\mathbf{v}_i$ is multiplied by its corresponding softmax score and summed to produce the self-attention layer output for the first sequence position, $\mathbf{b}_1$. Similar calculations for the remaining positions yield $\mathbf{a}_2$, $\mathbf{a}_3$, and $\mathbf{a}_4$. The complete self-attention matrix combines all $\mathbf{b}_i$ vectors. In practice, these computations are performed as matrix operations accelerated by GPUs.

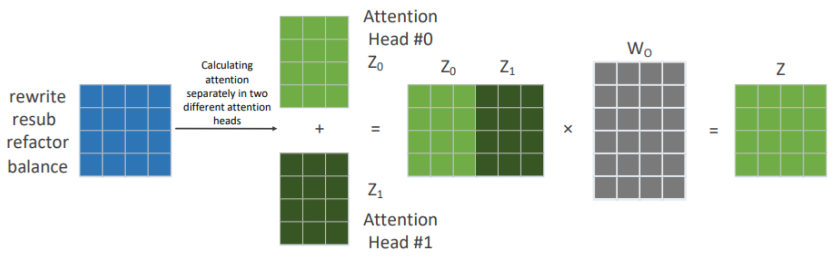


Figure 4: The process of computing multi-headed self-attention

To enhance the model's ability to focus on different positions, we implement a multi-head attention mechanism with two attention heads, each having independent query, key, and value weight matrices. As shown in Figure 4, each head independently performs the same operations described earlier to obtain its corresponding $\mathbf{b}_i$ vectors. These vectors form matrices $\mathbf{Z}_0$ and $\mathbf{Z}_1$, which are concatenated and multiplied by an additional weight matrix $\mathbf{W}_0$ to produce the final self-attention matrix $\mathbf{Z}$.

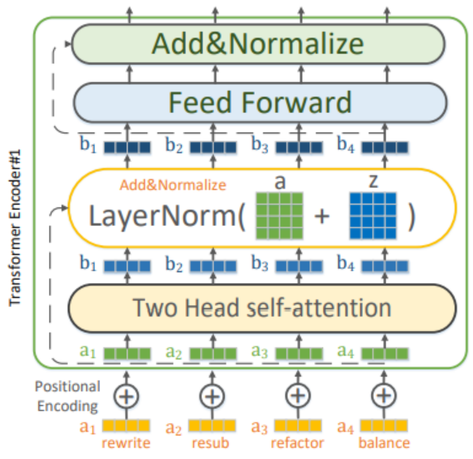


Figure 5: Transformer-based optimization sequence feature extractor.

b. Transformer Based Extraction
The complete transformer-based optimization sequence feature extractor is illustrated in Figure 5, showing its block structure.

For sequence feature extraction, we begin by converting the input optimization sequence into vectors $\mathbf{a}_i$ using the optimization sequence embedding method outlined in Section 4.1.1. These vectors are enhanced by adding the corresponding position encodings, creating updated vectors $\mathbf{a}_i$.

These enhanced vectors are then processed through the two-head self-attention module (detailed in Section 4.1.2.a), which generates the self-attention matrix $\mathbf{Z}$.

Following the attention computation, the output passes through a residual connection and layer normalization module. The residual connection combines the self-attention matrix $\mathbf{Z}$ with the original input vectors $\mathbf{a}_i$, which helps prevent network degradation during training. This combined output undergoes layer normalization as described in [17], which standardizes the activation values within that layer.

The normalized output is then transformed by a feed-forward neural network. After this transformation, another residual connection adds the feed-forward network's input to its output, followed by a second layer normalization to produce the final block output.

Our implementation stacks three of these blocks sequentially to form the complete optimization sequence feature extractor. The final output from this stack is further processed through a linear layer that performs dimensional scaling to achieve the desired feature representation.

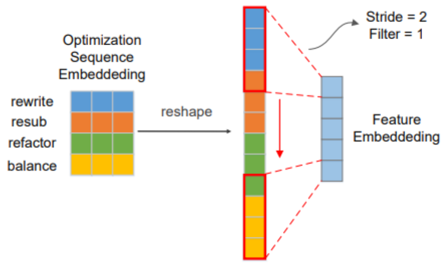


Figure 6: Extraction of optimization sequence features using CNN

4.1.3 Recipe Feature Extraction Using CNN And LSTM

For comparative analysis, we implement CNN and LSTM approaches previously used in [8] and [9] respectively. Figure 6 illustrates a CNN-based optimization sequence feature extractor. In this approach, the embedded vector representation of the optimization sequence is reshaped, and features are extracted using a convolution kernel that moves top-down in two-step increments. During each movement, the convolution kernel performs dot products with the embedded vectors within its window, with the sequence features comprising the results of these dot product operations.

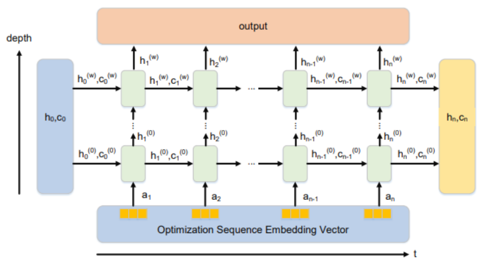


Figure 7: Feature extraction for optimization sequences using LSTM

Figure 7 demonstrates how LSTM extracts features from optimization sequences. The optimization sequence is processed as time-series data by the LSTM network, where each LSTM unit updates its cell state and hidden state output based on the current input and the previous unit's output. In our implementation, we use the final hidden state from each layer as the embedding feature representing the entire optimization sequence. Table 2 provides the detailed parameters for both CNN and LSTM implementations.

Figure 8 shows an AIG (And-Inverter Graph) example representing the logic function $F(a, b, c) = \neg(a \wedge b \vee b \wedge c)$, which illustrates how these circuits are structured before optimization.

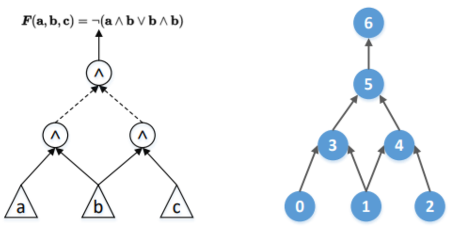


Figure 8: AIG Example of the Logic: $F(a, b, c) = \neg(a \wedge b \vee b \wedge c)$

4.2 Circuit Feature Extraction

Since the circuit can be represented as a directed acyclic graph, the graph neural network can be used to extract information about the structural features of the circuit. Figure 8 shows an example of AIG (8-a), which has three inputs and one output. The solid and dashed lines represent the buffer and inverter, respectively, and the node in the middle is the AND logic. In Figure 8-b, the AIG is converted to a graph representation. To distinguish the attributes of different nodes, different feature vectors are added to the nodes in the graph representation. For illustration, the first iteration of the simplified propagation rule will be shown below.

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a)

$$F = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 2 & 0 \\ 2 & 0 \\ 2 & 2 \\ 1 & 0 \end{bmatrix}$$

(b)

$$X = A \times F = \begin{bmatrix} 2 & 0 \\ 4 & 0 \\ 2 & 0 \\ 2 & 2 \\ 2 & 2 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

(c)

Figure 9: Matrix Operations and Results

Where A is the adjacency matrix representing the graph connectivity, and F is the feature matrix representing the node attributes. The feature addition rules follow [7]. Each node is initialized with a feature vector of length 2.

The first dimension of the feature matrix represents the node type, where the input, output, and AND gates are denoted as 0, 1, and 2, respectively. The second dimension is the number of inverters on the node inputs.

The matrix X obtained after multiplying A and F represents the node embeddings after propagation. For example, the row vector $[2, 0]$ in row 0 of X stands for the propagated feature of the corresponding node.

4.2.1 GCN Based Feature Extraction

A graph convolutional neural network, much like a regular convolution, creates convolutions by aggregating nodes and their neighbors to form new nodes. Like any other neural network, GCNs can be stacked with multiple layers. For each layer, the aggregation of nodes, also known as convolution, can be expressed as an equation:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}\right) \tag{6}$$

$\tilde{A}$ is the adjacency matrix with the addition of a self-loop, which allows each node to include its own features in the calculation. $\tilde{D}$ is the degree matrix of $\tilde{A}$, used to normalize nodes with larger degrees. $H^{(l)}$ denotes the node embedding at the output of layer l, which also serves as the input to the next layer. Initially, $H^{(0)} = F$, where F is the feature matrix of the node. $W^{(l)}$ represents the weight matrix of the l-th layer. σ is the activation function, e.g., ReLU.

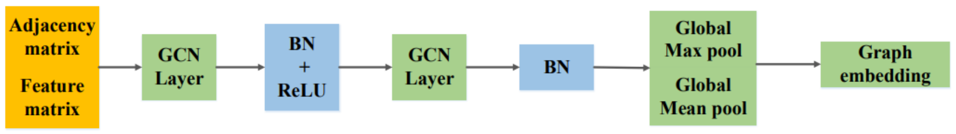


Figure 10: GCN-based circuit feature extractor.

The GCN-based circuit feature extractor used in our joint learning framework is illustrated in Figure10. First is the input, where each vector representing the node type is converted to a 1 × 3 vector by the embedding layer, and then is concatenated with a vector representing the number of inverters for each node input to become a 1 × 4 node feature vector. Two GCN layers are also included, enabling each node to learn node embeddings from their neighbors' neighbor nodes. Each GCN layer is followed by batch normalization (BN) which helps to improve the training speed and accuracy. The activation function ReLU is used to increase the nonlinear properties of the network. Finally, the graph embedding consists of global max pooling and global mean pooling of node embeddings. The final graph embeddings will be used as extracted circuit features for downstream QoR prediction tasks.

4.2.2 GAT-Based Feature Extraction

The weights on the edges of the GCN are fixed at the time of aggregation for each convolution. GAT introduces the attention mechanism to let the model learn the weight assignment. The weight learning formula is as follows:

$$\alpha_{ij} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\tilde{\mathbf{a}}^{\top}\left[\mathbf{W}\tilde{\mathbf{h}}_i \| \mathbf{W}\tilde{\mathbf{h}}_j\right]\right)\right)}{\sum_{k\in\mathcal{N}(i)}\exp\left(\mathrm{LeakyReLU}\left(\tilde{\mathbf{a}}^{\top}\left[\mathbf{W}\tilde{\mathbf{h}}_i \| \mathbf{W}\tilde{\mathbf{h}}_k\right]\right)\right)} \tag{7}$$

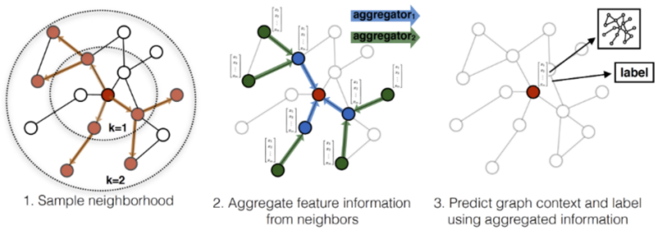

k=1
k=2
1. Sample neighborhood
aggregator1
aggregator2
2. Aggregate feature information from neighbors
label
3. Predict graph context and label using aggregated information

Figure 11: Graphsage schematic. Image cited from [15].

where α_{ij} denotes the attention coefficient between the i-th node and the j-th node, $\tilde{\mathbf{h}}_i$ is the node feature, and the other $\tilde{\mathbf{a}}^{\top}$, $\mathbf{W}$ are the learnable model parameters. The formulation first concatenates the representations of the target and source nodes and computes the correlation through the parameter network $\tilde{\mathbf{a}}^\top$. After that, the correlation is passed through the activation function LeakyReLU. Finally, the computed results are normalized by the softmax function to obtain the attention score α_{ij}.

Feature aggregation is performed according to the attention score with the following equation:

$$\tilde{\mathbf{h}}_i' = \sigma(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W \tilde{\mathbf{h}}_j) \tag{8}$$

The aggregation process is equivalent to weighted summation. The GAT-based circuit feature extractor is adapted from the module in Figure 10. Specifically, the GCN layer in it is replaced by the GAT layer, which means the aggregation process is changed to a weighted summation based on the attention coefficients. In addition, the activation function is changed from ReLU to ELU.

4.2.3 GraphSage-Based Feature Extraction

In GraphSAGE, the feature aggregation process is represented as a function. The (structural and feature) information of nodes is passed from point to point. Through the aggregation function, a node can aggregate the information of its neighbors and update the information of the current node through the update function (neural network). We used the MEAN aggregator with the following equation:

$$\mathbf{h}_v^{(k)} \leftarrow \sigma\left(\mathbf{W} \cdot \text{MEAN}\left(\{\mathbf{h}_v^{(k-1)}\} \cup \{\mathbf{h}_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)\right) \tag{9}$$

The mean aggregator concatenates the $(k-1)$-th layer vectors of the target and neighbor nodes, then operates to find the mean value for each dimension of the vector, and applies a nonlinear transformation to the obtained result to produce the node embedding for the k-th layer for the target node.

In addition, as shown in Figure 11, to save computational resources, GraphSAGE allows sampling a certain number of neighboring nodes for each node as the nodes to aggregate information from. In the GraphSAGE-based circuit feature extractor, the GCN layer in Figure 10 is replaced with GraphSAGE.

4.3 Joint Learning Policy

To predict the QoR of unseen circuit-optimization sequence pairs, optimization sequence feature extractor and circuit feature extractor are adopted as a joint policy. As shown in Figure 12, first, the optimization sequence and the AIG graph are passed into the sequence feature extractor and the circuit feature extractor, respectively.

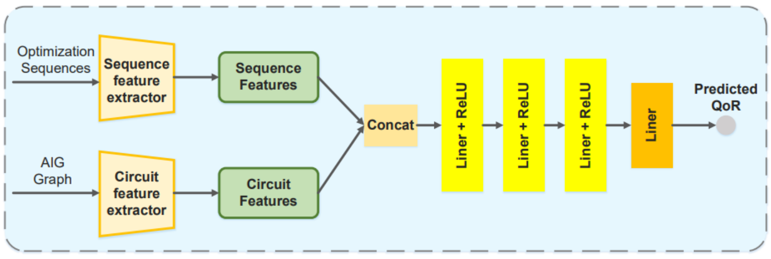


Figure 12: Sequence feature extractor and circuit feature extractor adopted a joint learning policy for predicting the QoR of optimization sequences

Table 1: Hyperparameters of the circuit feature extractor. In the GAT-based module, both GAT layers use two-head attention. Among them, the two-head attention of Layer1 is concatenated and the two-head attention of Layer2 is averaged.

GNN Type	**Input**	**Layer1**	**Layer2**	**Pool**	**Output**
GCN	4	64	64	Max+Mean	128
GAT	4	32×2	64	Max+Mean	128
GraphSage	4	64	64	Max+Mean	128

Table 2: Hyperparameters of the optimization sequence feature extractor. The dimension of batch size is omitted.

Model	**Hyperparameter**	**Value**
Transformer	Input	(20, 4)
	Num_Head	2
	Dim_feedforward	32
	Num_layers	3
	Linear	50
	Output	50
Sequence feature extractors (LSTM)	Input	(20, 3)
	Hidden_Size	64
	Num_layers	2
	Output	64
CNN	Input	60
	Filters	4
	Kernels	21, 24, 27, 30
	Stride	3
	Output	50

Table 3: Hyperparameters of the fully connected layer. The input dimensions are divided into two types, 178: Transformer/CNN + GNN, 192: LSTM + GNN.

FC Stack	**Input**	**Linear1**	**Linear2**	**Linear3**	**Linear4**	**Dropout**
	178/192	512	256	256	1	0.2

Then, the sequence features and circuit features output from the two extractors are concatenated together. The concatenated features are passed through three linear layers with the activation function ReLU. Finally, a linear layer outputs the predicted QoR. With such a structure, the neural network can learn features of both the optimization sequence and the circuit and link them together. This allows predictions to be passed from circuit to circuit rather than being restricted to a specific circuit. The detailed parameters of the network are presented in Tables 1, 2, and 3.

Table 4: Circuit characteristics of benchmarks. PI: Primary Inputs, PO: Primary Outputs, N: Number of gates, D: Depth.

Circuit	**Characteristics of Benchmarks**			
	PI	**PO**	**N**	**D**
aes_secworks	3087	2604	40778	41
tv80	636	361	11328	53
max	512	130	2865	287
apex1	45	45	1577	14
i2c	177	128	1169	14
k2	45	45	2289	22

Table 5: Experimental results of the joint model of circuit feature extractor and optimization sequence feature extractor for the QoR prediction task (the lower the better). Results are averaged over 3 runs with 3 different seeds.

Model	Transformer		CNN		LSTM	
	ACC/MAE	Epoch	ACC/MAE	Epoch	ACC/MAE	Epoch
GCN	0.176137451 ± 0.207634991	100	0.029959274 ± 0.047627506	100	0.032825145 ± 0.048436526	100
GAT	0.061311653 ± 0.04583693	100	0.037022231 ± 0.03948176	100	0.076249806 ± 0.076065366	100
GraphSage	0.031018462 ± 0.03968295	100	0.034120495 ± 0.040919589	100	0.017043361 ± 0.20489387	100

5 Approach

The architecture of our system consists of three main components: the graph encoder,the recipe encoder,and the fusion mechanism coupled with the decoder. In this work, we focused on improving the decoding process and introduced modifications to other components of the model to enhance overall performance.These aspects are detailed below, emphasizing the specific contributions of our approach within each block.

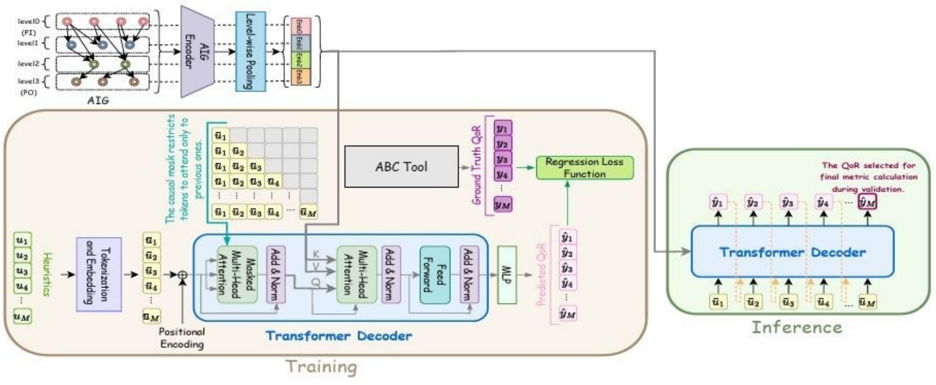


Figure 13: GraphMT architecture during training and inference modes

5.1 Self-Supervised Learning Auxiliary Task

As an auxiliary task helping the model during training, we train the model to predict the trajectory of QoR evolution. To achieve this, the model step by step predicts all M intermediate QoRs in a causal manner. For predicting the i-th QoR, the model takes as input all heuristics before the i-th step, $(r_1, r_2, \ldots, r_i)$. This predictive SSL task by itself is aligned with the architecture of our transformer decoder to help the model benefit from extra information within intermediate steps.

5.2 Graph Encoder

In the AIG segment, we employ a graph encoder E_θ based on the following formulation:

$$H = E_\theta(G) \tag{10}$$

Here, $H = \{h_1, h_2, ..., h_N\}$ represents the set of node embeddings and $h_i \in \mathbb{R}^{d_h}$ is i-th node embedding. Similar to the approach used in OpenABC [28], we utilize a Graph Convolutional Neural Network (GCN) with the same node embedding approach to encode the AIGs.

5.3 Level-wise Node Pooling

Although the GCNs can encode graphs properly, they are unable to particularly incorporate the inductive bias of DAGs. To benefit from this bias to improve the graph representation learner, we have used level-wise pooling mechanism. The depth of a node v is calculated through the function depth(v) because of the partial order intrinsic to the DAG. The set of node embeddings can be partitioned based on the depth of each node as follows:

$$H = \{H_0, H_1, ..., H_D\} \tag{11}$$

where H_l represents the set of embeddings for nodes located at l-th level of AIG. Finally, the AIG graph is represented as a sequence for the decoder layer described in the next section. The sequence is derived according to the combination of mean and max poolings as follows:

$$\bar{H} = (\bar{h}_0, \bar{h}_1, ..., \bar{h}_D), \quad \bar{h}_l = \text{POOL}_{\text{level}}(H_l) \tag{12}$$

where $\bar{H}$ is the sequence of representation of DAG and $\bar{h}_l \in \mathbb{R}^{2\times d_h}$ is the embedding of l-th level of AIG. The $\text{POOL}_{\text{level}}$ is the concatenation of mean and max poolings.

5.4 Recipe Embedding

Each recipe, denoted as $r_i = (u_1, u_2, ..., u_M)$, where each $u_j \in T$ represents the j-th AIG operation within the recipe, is tokenized using a one-hot vector of length C. A lookup table is utilized to generate learned embeddings to convert the input tokens to vectors $\bar{u}_i \in \mathbb{R}^{2\times d_h}$ within the sequence of embeddings $\bar{r}_i = (\bar{u}_1, \bar{u}_2, ..., \bar{u}_M)$.

5.5 Transformer Decoder

Once the recipes are embedded and the AIG is converted to a sequence, inspired by Machine Translation models [27], we have used a sequence-to-sequence alignment module using only transformer decoder module to fuse the AIG sequence with heuristics' embeddings. In other words, this transformer module maps AIG sequence and heuristics' embeddings to the trajectory of QoRs in causal manner. More specifically:

$$\hat{Y} = \text{Transformer}(\bar{H}, \bar{r}_i) \tag{13}$$

$$\hat{Y} = (\hat{y}_1, ..., \hat{y}_M) \tag{14}$$

where $\hat{Y}$ consists of the QoRs of the trajectory. Our transformer decoder containing 6 main components is designed and formulated as follows:

Positional Encoding: Sine/Cosine positional encoding (PE) is added to the sequence of heuristics embeddings element-wise as follows:

$$PE(m, 2k) = \sin\left(\frac{m}{10000^{k/d_h}}\right) \tag{15}$$

$$PE(m, 2k+1) = \cos\left(\frac{m}{10000^{k/d_h}}\right) \tag{16}$$

$$z_i = \bar{r}_i + PE \tag{17}$$

where z_i is the sequence of positionally encoded embeddings of Optimizers.

Masked Multi-Head Self-Attention: If $Q = z_i W_q \in \mathbb{R}^{M\times 2d_h}$, $K = z_i W_k \in \mathbb{R}^{M\times 2d_h}$, and $V = z_i W_v \in \mathbb{R}^{M\times 2d_h}$ are the query, key, and value matrices, the contextualized heuristics' embedding ($\bar{z}_i$) is derived as follows:

$$\bar{z}_i = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + \bar{M}\right) V \tag{18}$$

The mask $\bar{M}$ is typically an upper triangular matrix with zeros on and below the diagonal and $-\infty$ above the diagonal for each sequence in the batch. This structure effectively imposes causality and ensures that each position in the sequence can only attend to itself and previous positions, not to any future positions.

Cross-Attention: This module computes the attention weights between the element of AIG sequence and contextualized recipe' embeddings. Finally, a Feed-Forward layer (FFN) with ReLU function is applied element-wise to the sequence.

$$\bar{h}^i = \text{FFN}(\text{MultiHeadAttention}(\bar{z}_i, \bar{H}, \bar{H})) \tag{19}$$

where $\bar{h}^i = (u_1, u_2, ..., u_M)$ is the output sequence of embeddings for input AIG and i-th recipe; $u_i \in \mathbb{R}^{2\times d_h}$ represents generated embedding corresponding to i-th QoR.

Add & Norm: Each sub-layer (self-attention and feed-forward network) includes a residual connection followed by layer normalization:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{20}$$

MLP Regressor: Once the embeddings for each QoR has been generated sequentially, we use a MLP module to map the embeddings to final QoR as follows:

$$\hat{y}_i = \text{MLP}(\bar{h}^i) \tag{21}$$

5.6 Fine-Tuning

The integration of ControlNet like architecture with causal transformers presents a novel approach to fine-tuning pre-trained language models while preserving their original capabilities. This methodology employs two key mechanisms: a locked backbone with a trainable copy, and zero-initialized projection layers for gradual parameter growth.

5.6.1 Locked Backbone with Trainable Copy

We begin by creating a frozen copy of a pre-trained causal transformer model $\mathcal{M}_{\text{pretrained}}$ with parameters $\theta_{\text{pretrained}}$, alongside a trainable copy $\mathcal{M}_{\text{trainable}}$ with parameters $\theta_{\text{trainable}}$ initialized identically to $\theta_{\text{pretrained}}$. This dual-model structure is formulated as:

$$\theta_{\text{trainable}} \leftarrow \theta_{\text{pretrained}} \tag{22}$$
$$\nabla_{\theta_{\text{pretrained}}}\mathcal{L} = 0 \tag{23}$$

where $\mathcal{L}$ represents the loss function. The gradient constraint $\nabla_{\theta_{\text{pretrained}}}\mathcal{L} = 0$ ensures that the original model's parameters remain fixed throughout the fine-tuning process, preserving the knowledge encoded in the pre-trained model.

For a causal transformer with L layers, each layer $l \in 1, 2, ..., L$ consists of self-attention blocks and feed-forward networks. The locked backbone preserves these complex interactions while the trainable copy learns task-specific adaptations.

5.6.2 Zero-Initialized Projection Layers

To connect the trainable copy to the original model, we introduce zero-initialized linear projection layers $\mathcal{Z}$in and $\mathcal{Z}$out at strategic points in the architecture. For a hidden state $\mathbf{h}^l$ at layer l, the connection is established as:

$$\mathbf{h}^l_{\text{combined}} = \mathbf{h}^l_{\text{pretrained}} + \mathcal{Z}^l_{\text{out}}(\mathcal{M}^l_{\text{trainable}}(\mathbf{h}^l_{\text{pretrained}} + \mathcal{Z}^l_{\text{in}}(\mathbf{c}))) \tag{24}$$

where $\mathbf{c}$ represents the conditioning information and $\mathcal{Z}\text{in}^l, \mathcal{Z}\text{out}^l$ are defined as:

$$\mathcal{Z}_{\text{in}}^{l}(\mathbf{x}) = \mathbf{W}_{\text{in}}^{l}\mathbf{x} + \mathbf{b}_{\text{in}}^{l} \tag{25}$$
$$\mathcal{Z}\text{out}^{l}(\mathbf{x}) = \mathbf{W}_{\text{out}}^{l}\mathbf{x} + \mathbf{b}_{\text{out}}^{l} \tag{26}$$

with initialization:

$$\mathbf{W}_{\text{in}}^{l}, \mathbf{W}_{\text{out}}^{l} \leftarrow \mathbf{0} \tag{27}$$
$$\mathbf{b}_{\text{in}}^{l}, \mathbf{b}_{\text{out}}^{l} \leftarrow \mathbf{0} \tag{28}$$

This zero-initialization ensures that at the beginning of training:

$$\mathbf{h}_{\text{combined}}^{l} = \mathbf{h}_{\text{pretrained}}^{l} \tag{29}$$

The gradual parameter growth occurs as training progresses, with the weights of $\mathcal{Z}_{\text{in}}^{l}$ and $\mathcal{Z}_{\text{out}}^{l}$ evolving from their zero initialization according to:

$$\mathbf{W}_{\text{in/out}}^{l}(t) = \mathbf{W}_{\text{in/out}}^{l}(t-1) - \eta\nabla_{\mathbf{W}_{\text{in/out}}^{l}}\mathcal{L} \tag{30}$$
$$\mathbf{b}_{\text{in/out}}^{l}(t) = \mathbf{b}_{\text{in/out}}^{l}(t-1) - \eta\nabla\mathbf{b}_{\text{in/out}}^{l}\mathcal{L} \tag{31}$$

where t represents the training step and η is the learning rate.

5.6.3 Implementation in Causal Transformers

For causal transformers specifically, we implement this architecture by connecting the zero-initialized projection layers to the output of each transformer block. Given a sequence of tokens $\mathbf{X} = [x_1, x_2, ..., x_n]$, the forward pass through the modified architecture is:

$$\mathbf{H}^{0} = \text{Embedding}(\mathbf{X}) \tag{32}$$

For each layer $l \in 1, 2, ..., L$:

$$\mathbf{H}_{\text{pretrained}}^{l} = \text{TransformerBlock}_{\text{pretrained}}^{l}(\mathbf{H}_{\text{pretrained}}^{l-1}) \tag{33}$$
$$\mathbf{H}_{\text{trainable}}^{l} = \text{TransformerBlock}_{\text{trainable}}^{l}(\mathbf{H}_{\text{pretrained}}^{l-1} + \mathcal{Z}_{\text{in}}^{l}(\mathbf{c})) \tag{34}$$
$$\mathbf{H}_{\text{combined}}^{l} = \mathbf{H}_{\text{pretrained}}^{l} + \mathcal{Z}_{\text{out}}^{l}(\mathbf{H}_{\text{trainable}}^{l}) \tag{35}$$

The final output is computed as:

$$\mathbf{Y} = \text{OutputLayer}(\mathbf{H}_{\text{combined}}^{L}) \tag{36}$$

6 Results

The training is performed on a 2× Intel Xeon Silver 4214R, 130GB RAM, and Nvidia A100. The loss function used during training is **Mean Square Error (MSE)**, with the Adam optimizer, a learning rate of 0.001, batch size of 1, and a total of 10 epochs. The experiments are implemented in Python 3.12 using the deep learning framework PyTorch and the graph neural network framework PyTorch-Geometric. The data used for training and testing is generated by running optimization recipes on the ABC synthesis tool. Specifically, the dataset comprises of 3 benchmark circuits for training, and 3 different benchmark circuits for fine-tuning. The characteristics of these circuits are summarized in Table 4. Each circuit is optimized using

$K = 1500$ different optimization sequences, each of length $L = 20$. This results in a total dataset size of 9000 examples, where the label corresponds to the number of levels remaining after optimization. In the case of GraphMT, the dataset comprises 4500 examples, with each example representing a trajectory of levels throughout the optimization process.

A **joint learning model** is trained using 80% of the optimization sequences across the 3 circuits. The model is then tested on unseen recipe. From the training set, 10% is reserved for validation. Table 5 reports the results for nine combinations of sequence and circuit feature extractors under the joint learning policy. The combination of a **Transformer and GraphSAGE** achieves the best performance with a mean absolute error (MAE) of **0.031018462 ± 0.03968295** . This demonstrates the advantage of the self-attention mechanism in capturing sequence features and the effectiveness of learned aggregation functions for graph-based circuit features.

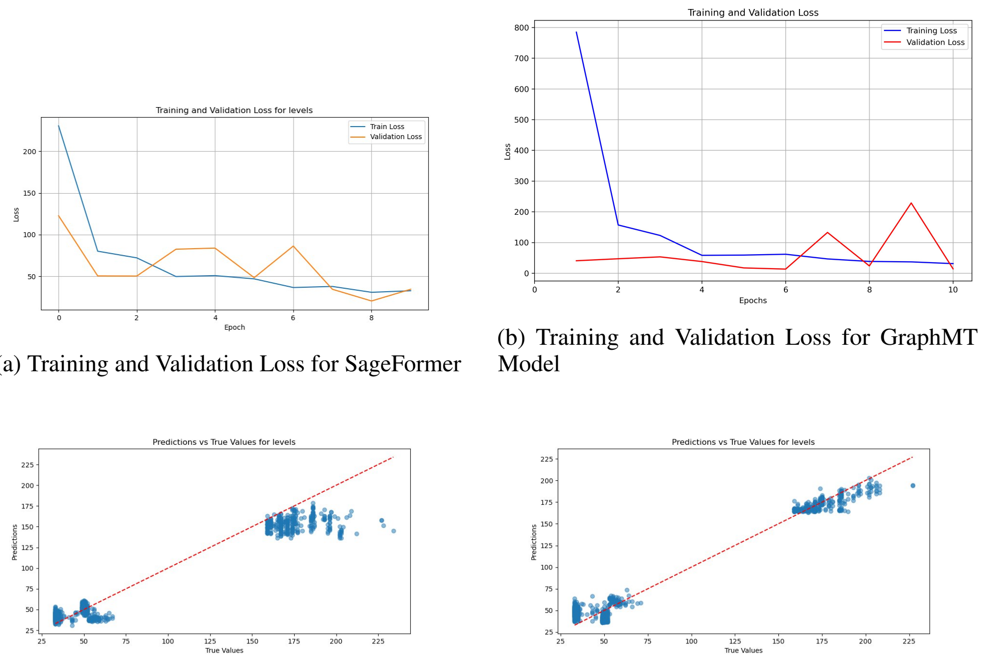
In contrast to conventional graph tasks, the **GAT model** yielded the poor results. We hypothesize that the varying attention weights assigned to neighbor nodes in GAT are not effective for AIG representations, as the importance of neighbors may depend heavily on whether the input is a buffer or an inverter.

We also observe that the **LSTM-based** sequence extractor performs slightly worse than the Transformer but benefits from reduced training time. This is likely because LSTM is a lighter model compared to the Transformer, and our current dataset lacks large-scale industrial circuits and comprehensive feature coverage. Thus, LSTM may be more appropriate for lightweight use cases. However, the Transformer has a higher upper bound in terms of performance, and we plan to further explore its potential using more extensive datasets in the future.

Figure 14a shows the training and validation loss curves, indicating the model's learning progress over epochs. The training converges quickly, with both training and validation losses stabilizing around the 10th epoch. This demonstrates the efficiency of the joint learning framework in capturing both circuit structure and optimization sequence features. Figure 14c presents the predicted and true QoR values for unseen circuits and unseen optimization sequences **without fine-tuning.** The results suggest that the model has difficulty generalizing in completely new scenarios, as the predicted values deviate significantly from the ground truth.

In contrast, Figure 14d displays the predicted and true QoR values for unseen circuits and unseen optimization sequences **after applying fine-tuning.** The model's predictions align more closely with the actual QoR values, highlighting the effectiveness of **fine-tuning** in enhancing performance on previously unseen data. This demonstrates the importance of incorporating adaptive learning strategies for real-world deployment across diverse circuit designs and optimization patterns.

The **GraphMT model** showed effective learning behavior over the course of training. It exhibited a rapid decline in training loss, and the validation loss generally improved, indicating good generalization. Although there were some fluctuations in the validation loss between epochs 6 and 9, the model stabilized in the final epoch. Ultimately, both training and validation losses were low and closely aligned, demonstrating strong convergence and minimal overfitting.(see Figure 14b)



(a) Training and Validation Loss for SageFormer

(b) Training and Validation Loss for GraphMT Model

(c) GraphMT - without Fine Tuning

(d) GraphMT - with Fine Tuning

Figure 14: :Results

QoR Predictor-Based Synthesis Recipe Selection

To identify an effective synthesis recipe for a given design, we employed Particle Swarm Optimization (PSO) in conjunction with GraphMT, both with and without fine-tuning. The QoR predictor was used to evaluate the potential quality of each synthesized design during the search process. The results, presented in Table 6, show that GraphMT with fine-tuning is capable of achieving results close to the optimal quality of results (QoR) defined by ABC. In contrast, GraphMT without fine-tuning consistently yielded suboptimal outcomes, indicating that the lack of task-specific adaptation limits its effectiveness. Throughout this evaluation, ABC has been treated as the baseline for optimal QoR, serving as a reference for comparison.

	ABC			**GraphMT (W/O Finetune)**			**GraphMT (with Finetune)**		
Benchmark	**i/o**	**Nodes**	**Levels**	**i/o**	**Nodes**	**Levels**	**i/o**	**Nodes**	**Levels**
max	512/130	2827	159	512/130	2858	196	512/130	2833	161
tv80	636/361	8802	49	636/361	9031	52	636/361	9007	49
aes_secworks	3087/2604	30618	33	3087/2604	30707	43	3087/2604	31573	33
i2c	177/128	900	13	177/128	872	18	177/128	1464	13
k2	45/45	1465	12	45/45	1563	21	45/45	1461	13
apex1	45/45	1099	12	45/45	1153	16	45/45	1163	12

Table 6: Performance comparison of ABC vs GraphMT with and without fine-tuning

7 Conclusion

This work presents a comprehensive study on QoR prediction in Logic Synthesis Optimization (LSO), culminating in an architecture that combines GraphSage-based level-wise AIG encoding with Transformer-based recipe modeling. Our experiments on nine model variants revealed that Transformer models excel at capturing sequential dependencies in optimization recipes, while heirarchial GraphSage pooling effectively captures topological circuit features. The integration of a ControlNet-inspired fine-tuning strategy further enhanced generalization to unseen circuits. Together, these innovations achieved state-of-the-art performance on benchmark datasets, offering a human-like and scalable approach to QoR trajectory modeling in LSO.

Future directions include improving the scalability of graph pooling methods to better capture circuit semantics and integrating our approach with downstream EDA stages such as placement and routing for end-to-end optimization. Given the limited availability of diverse circuit datasets, incorporating few-shot learning and LLM-based prompting techniques could enable better generalization from limited samples. Additionally, exploring LLM-enhanced architectures for generating or refining optimization sequences may unlock new synergies between pretrained language models and logic synthesis workflows. Finally, expanding our framework to predict multiple QoR metrics simultaneously would enhance its applicability to real-world design constraints.

References

[1] E. Testa, M. Soeken, L. G. Amar, et al., "Logic synthesis for established and emerging computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 165–184, 2018.

[2] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. Int. Conf. on Computer Aided Verification*, Springer, 2010, pp. 24–40.

[3] A. B. Kahng, "New directions for learning-based IC design tools and methodologies," in *Proc. ASP-DAC*, IEEE, 2018, pp. 405–410.

[4] W. Haaswijk, E. Collins, B. Seguin, et al., "Deep learning for logic optimization algorithms," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2018, pp. 1–4.

[5] K. Zhu, M. Liu, H. Chen, et al., "Exploring Logic Optimizations with Reinforcement Learning and Graph Convolutional Network," in *Proc. MLCAD*, IEEE, 2020, pp. 145–150.

[6] A. Hosny, S. Hashemi, M. Shalan, et al., "Drills: Deep reinforcement learning for logic synthesis," in *Proc. ASP-DAC*, IEEE, 2020, pp. 581–586.

[7] A. B. Chowdhury, B. Tan, R. Karri, et al., "OpenABC-D: A Large-Scale Dataset For Machine Learning Guided Integrated Circuit Synthesis," *arXiv preprint arXiv:2110.11292*, 2021.

[8] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. 55th Annual Design Automation Conference (DAC)*, 2018, pp. 1–6.

[9] C. Yu and W. Zhou, "Decision Making in Synthesis cross Technologies using LSTMs and Transfer Learning," in *Proc. MLCAD*, 2020, pp. 55–60.

[10] Y. LeCun, B. Boser, J. S. Denker, et al., "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] A. Vaswani, N. Shazeer, N. Parmar, et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[13] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[14] P. Veličković, G. Cucurull, A. Casanova, et al., "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[15] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. NeurIPS*, 2017, pp. 1025–1035.

[16] J. Devlin, M. W. Chang, K. Lee, et al., "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] A. Paszke, S. Gross, F. Massa, et al., "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.

[20] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[21] OpenCores, "OpenCores hardware RTL designs." Available: `https://opencores.org/`

[22] MIT Lincoln Laboratory, "MIT Common Evaluation Platform (CEP)." Available: `https://github.com/mit-ll/CEP`

[23] X. J. Jiang, "AES 128/256-bit symmetric block cipher." Available: `https://github.com/crypt-xie/XCryptCore/tree/master/ciphers/aes`

[24] J. Strömbergson and O. Kindgren, "AES 128/256-bit symmetric block cipher." Available: `https://github.com/secworks/aes`

[25] J. Balkind, M. McKeown, Y. Fu, et al., "OpenPiton: An open source manycore research framework," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 217–232, 2016.

[26] T. Ajayi and D. Blaauw, "OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain," in *Proc. Government Microcircuit Applications and Critical Technology Conference (GOMACTech)*, 2019.

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. A., Kaiser, Ł., Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (pp. 5998-6008).

[28] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. OpenABC-D: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv preprint arXiv:2110.11292*, 2021.

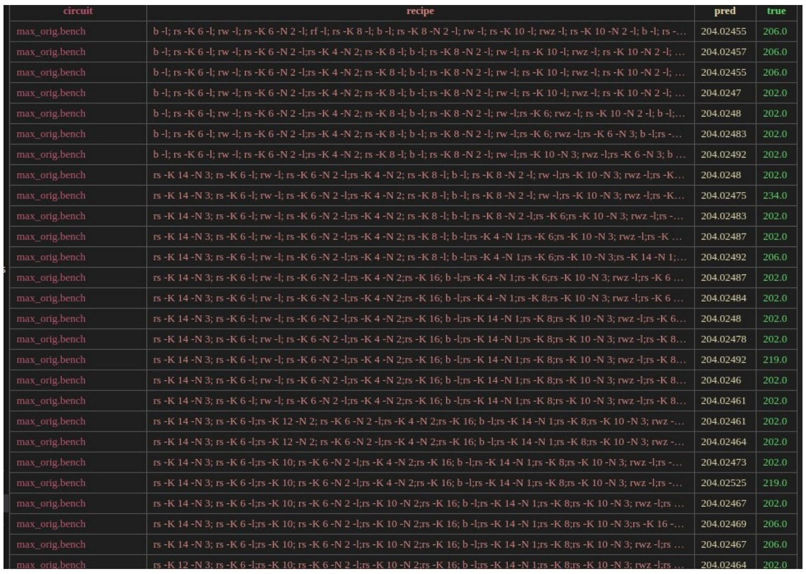
[29] Yuankai Luo, Veronika Thost, and Lei Shi. Transformers over Directed Acyclic Graphs. *Advances in Neural Information Processing Systems*, 36, 2024.

[30] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[31] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

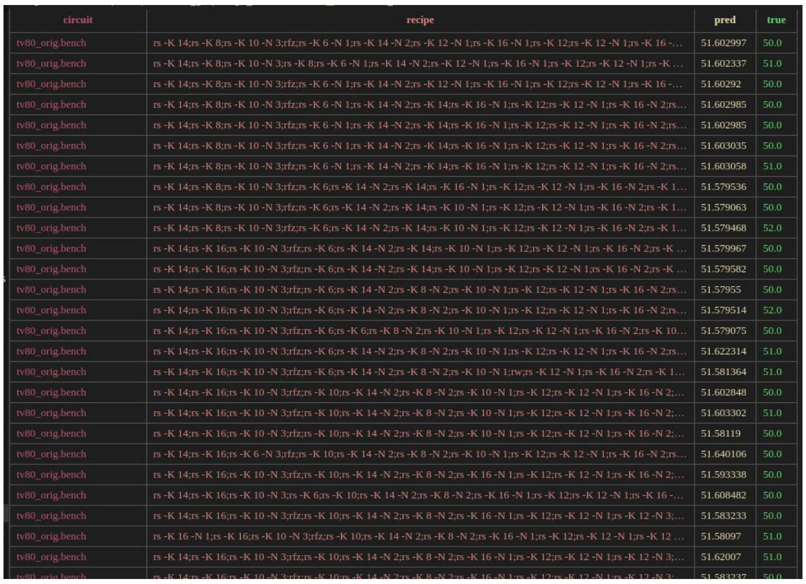
[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

8 Predicted Result



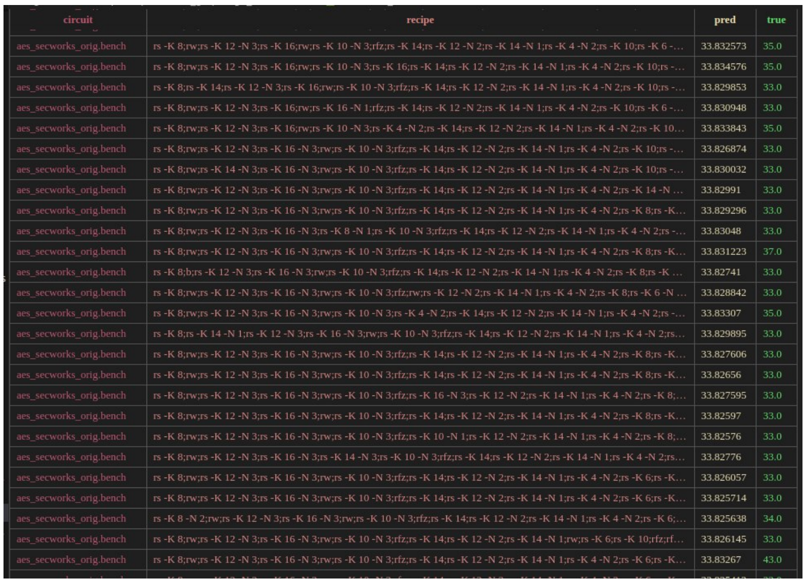
circuit	recipe	pred	true
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l; rf -l; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l; rs -K 10 -l; rwz -l; rs -K 10 -N 2 -l; b -l; rs -…	204.02455	206.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l; rs -K 10 -l; rwz -l; rs -K 10 -N 2 -l; …	204.02457	206.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l; rs -K 10 -l; rwz -l; rs -K 10 -N 2 -l; …	204.02455	206.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l; rs -K 10 -l; rwz -l; rs -K 10 -N 2 -l; …	204.0247	202.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l;rs -K 6; rwz -l; rs -K 10 -N 2 -l; b -l;…	204.0248	202.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l;rs -K 6; rwz -l;rs -K 6 -N 3; b -l;rs -…	204.02483	202.0
max_orig.bench	b -l; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l;rs -K 10 -N 3; rwz -l;rs -K 6 -N 3; b …	204.02492	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l;rs -K 10 -N 3; rwz -l;rs -K…	204.0248	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l; rw -l;rs -K 10 -N 3; rwz -l;rs -K…	204.02475	234.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l; rs -K 8 -N 2 -l;rs -K 6;rs -K 10 -N 3; rwz -l;rs -…	204.02483	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l;rs -K 4 -N 1;rs -K 6;rs -K 10 -N 3; rwz -l;rs -K …	204.02487	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2; rs -K 8 -l; b -l;rs -K 4 -N 1;rs -K 6;rs -K 10 -N 3;rs -K 14 -N 1;…	204.02492	206.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 4 -N 1;rs -K 6;rs -K 10 -N 3; rwz -l;rs -K 6 …	204.02487	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 4 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 6 …	204.02484	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 6…	204.0248	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 8…	204.02478	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 8…	204.02492	219.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 8…	204.0246	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l; rw -l; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -K 8…	204.02461	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 12 -N 2; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -…	204.02461	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 12 -N 2; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -…	204.02464	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -…	204.02473	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 4 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs -…	204.02525	219.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 10 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs …	204.02467	202.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 10 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3;rs -K 16 -…	204.02469	206.0
max_orig.bench	rs -K 14 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 10 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs …	204.02467	206.0
max_orig.bench	rs -K 12 -N 3; rs -K 6 -l;rs -K 10; rs -K 6 -N 2 -l;rs -K 10 -N 2;rs -K 16; b -l;rs -K 14 -N 1;rs -K 8;rs -K 10 -N 3; rwz -l;rs …	204.02464	202.0

Figure 15: Predicted Result CSV snapshot max_orig



circuit	recipe	pred	true
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 12 -N 1;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -…	51.602997	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rs -K 8;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 12 -N 1;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K …	51.602337	51.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 12 -N 1;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -…	51.60292	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 14;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.602985	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 14;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.602985	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 14;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.603035	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6 -N 1;rs -K 14 -N 2;rs -K 14;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.603058	51.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 14;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K 1…	51.579536	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 14;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K 1…	51.579063	50.0
tv80_orig.bench	rs -K 14;rs -K 8;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 14;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K 1…	51.579468	52.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 14;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K …	51.579967	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 14;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K …	51.579582	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.57955	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.579514	52.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 6;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs -K 10…	51.579075	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.622314	51.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 6;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rw;rs -K 12 -N 1;rs -K 16 -N 2;rs -K 1…	51.581364	51.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;…	51.602848	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;…	51.603302	51.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;…	51.58119	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 6 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 10 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;rs…	51.640106	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -N 2;…	51.593338	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rs -K 6;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 16 -…	51.608482	50.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 12 -N 3;…	51.583233	50.0
tv80_orig.bench	rs -K 16 -N 1;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 12 …	51.58097	51.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 12 -N 3;…	51.62007	51.0
tv80_orig.bench	rs -K 14;rs -K 16;rs -K 10 -N 3;rfz;rs -K 10;rs -K 14 -N 2;rs -K 8 -N 2;rs -K 16 -N 1;rs -K 12;rs -K 12 -N 1;rs -K 12 -N 3;	51.583237	50.0

Figure 16: Predicted Result CSV snapshot tv80



circuit	recipe	pred	true
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -K 6 -…	33.832573	35.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16;rw;rs -K 10 -N 3;rs -K 16;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -…	33.834576	35.0
aes_secworks_orig.bench	rs -K 8;rs -K 14;rs -K 12 -N 3;rs -K 16;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -…	33.829853	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16;rw;rs -K 16 -N 1;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -K 6 -…	33.830948	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16;rw;rs -K 10 -N 3;rs -K 4 -N 2;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10…	33.833843	35.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -…	33.826874	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 14 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 10;rs -…	33.830032	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 14 -N …	33.82991	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K…	33.829296	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rs -K 8 -N 1;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -…	33.83048	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K…	33.831223	37.0
aes_secworks_orig.bench	rs -K 8;b;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K …	33.82741	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rw;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K 6 -N …	33.828842	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rs -K 4 -N 2;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -…	33.83307	35.0
aes_secworks_orig.bench	rs -K 8;rs -K 14 -N 1;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs…	33.829895	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K…	33.827606	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K…	33.82656	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 16 -N 3;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;…	33.827595	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;rs -K…	33.82597	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 10 -N 1;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 8;…	33.82576	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rs -K 14 -N 3;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs…	33.82776	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 6;rs -K…	33.826057	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 6;rs -K…	33.825714	33.0
aes_secworks_orig.bench	rs -K 8 -N 2;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 6;…	33.825638	34.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rw;rs -K 6;rs -K 10;rfz;rf…	33.826145	33.0
aes_secworks_orig.bench	rs -K 8;rw;rs -K 12 -N 3;rs -K 16 -N 3;rw;rs -K 10 -N 3;rfz;rs -K 14;rs -K 12 -N 2;rs -K 14 -N 1;rs -K 4 -N 2;rs -K 6;rs -K…	33.83267	43.0

Figure 17: Predicted Result CSV snapshot aes_secworks