A Hybrid Reinforcement Learning Framework for Efficient Physical Design Parameter Tuning

HAO-HSIANG HSIAO, Georgia Institute of Technology, USA
YI-CHEN LU, NVIDIA Research, USA
PRUEK VANNA-IAMPIKUL, Burapha University, Thailand
SUNG KYU LIM, Georgia Institute of Technology, USA

Traditional Design Space Exploration (DSE) methods in Physical Design (PD), such as Bayesian Optimization (BO) and Ant Colony Optimization (ACO), as well as state-of-the-art commercial tools like Synopsys DSO.ai, typically treat the design flow as a black box, lacking insight into the underlying designs. This hinders their ability to generalize across unseen designs. In this paper, we introduce FastTuner, an innovative Reinforcement Learning (RL) agent that leverages Graph Neural Networks (GNNs) and Transformers to understand the underlying designs and enable rapid DSE on unseen designs across various PD stages. Our approach incorporates an attention-based framework for autoregressive and conditional parameter tuning and introduces a power, performance and area (PPA) estimator to predict end-of-flow PPA metrics, significantly accelerating RL reward computation. Extensive evaluations on seven industrial designs using the TSMC 28nm technology node demonstrate that FastTuner significantly outperforms existing state-of-the-art DSE techniques in both optimization quality and runtime, achieving improvements of up to 79.38% in Total Negative Slack (TNS), 12.22% in total power, and more than 50x reduction in runtime.

CCS Concepts: • **Hardware → Physical design (EDA)**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Physical Design, Reinforcement Learning, graph neural network

1 INTRODUCTION

Commercial Physical Design (PD) tools offer a broad range of tunable parameters, enabling customization to meet diverse design requirements. While this flexibility enhances optimization potential, it also complicates design space exploration, making it difficult to pinpoint configurations that optimize design goals. Conventional methods, including the widely used "auto-tuning" features in commercial tools, often suffer from prohibitively long runtimes and a vast number of permutations, stretching over days to weeks. Moreover, these methods require a "cold start" for each new design and struggle to leverage knowledge from previous tuning efforts. As design complexity escalates and the demand for rapid time-to-market intensifies, traditional approaches are becoming increasingly impractical. Consequently, there is an urgent need for a more efficient and scalable Design Space Exploration (DSE) technique.

Recent advancements in automated DSE techniques have emerged to tackle the challenges of extensive PD design spaces. In the industry, tools like Synopsys DSO.ai [1] and Cadence Cerebrus Intelligent Chip Explorer [2] are designed to automatically optimize engineer-specified design goals. In academia, state-of-the-art methods, primarily based on Bayesian techniques, have been developed to optimize Power, Performance, and Area (PPA) objectives, as demonstrated in [3–6]. Recently, more advanced techniques, such as those proposed in [7] (which is publicly available

Authors' addresses: Hao-Hsiang Hsiao, Georgia Institute of Technology, Atlanta, GA, USA, thsiao@gatech.edu; Yi-Chen Lu, NVIDIA Research, USA, yilu@nvidia.com; Pruek Vanna-Iampikul, Burapha University, Chonburi, Thailand, pruek.va@eng.buu.ac.th; Sung Kyu Lim, Georgia Institute of Technology, Atlanta, Georgia, USA, limsk@ece.gatech.edu.

2025. XXXX-XXXX/2025/9-ART $15.00
https://doi.org/

as an open-source implementation[1]) and [8], have been developed to effectively address multi-objective optimization. While these methods have shown considerable effectiveness, they fall short in leveraging underlying design features, requiring time-consuming and resource-intensive re-tuning for each new design.

To address the limitations of existing methods, some recent works have attempted to embed design features and demonstrate generalizability across different designs. For example, [9] leverages a generative adversarial network (GAN) to optimize clock tree synthesis, while [10] introduces a deep reinforcement learning approach for optimizing placement parameters. However, both of these approaches are confined to specific stages within the PD workflow and are not fully "automatic." They require substantial expertise to pre-configure near-optimal solutions, limiting their ability to achieve optimal performance without manual intervention.

Several other works, both in academia and industry [11–15], have been proposed to address the design space exploration problem. However, these methods often face limitations in transferability and scalability and typically rely on static models or datasets. Many of these studies focus on niche architectures or specific environments, such as the RISC-V BOOM microarchitecture or FPGA designs, which may restrict the broader applicability of their findings to other architectures or design spaces. These methods can struggle with scalability, particularly when applied to larger or more complex design spaces, and may require significant retraining or adaptation when design constraints or objectives change. Additionally, the reliance on static datasets can lead to suboptimal results compared to online methods that adapt in real-time to evolving design parameters and environments. Consequently, the effectiveness of these techniques often depends on the representativeness of the training data, which might not always reflect real-world scenarios, further limiting their practical utility.

In this paper, we introduce a fast and fully automatic RL tuning methodology using Decision Transformers to address current limitations[16–18]. Our framework features three key components that accelerate the exploration process: (1) Offline-trained PPA estimators provide immediate feedback to the RL agent, effectively replacing the time-consuming place-and-route (P&R) process. (2) Our framework generalizes effectively to unseen designs encompassed by the spectrum of prior training, delivering rapid adaptation without per-design retraining. (3) The FastTuner is built on a Transformer Decoder framework, allowing users to fine-tune specific parameter subsets and efficiently reduce the search space.

Our primary contributions include:

- Hybrid FastTuner Framework: We present a novel Hybrid FastTuner framework that enables online RL tuning using offline-trained PPA estimators, effectively eliminating the need for the extensive P&R process and reducing tuning duration from hours to seconds. This hybrid approach ensures rapid tuning by leveraging pre-trained models to provide instant feedback, allowing real-time and iterative parameter tuning.
- Generalization Through GNN: FastTuner supports transfer learning through a GNN, enabling it to adapt to diverse designs. Beyond simply capturing correlations between parameters and PPA metrics, our approach leverages underlying design features to drive optimization. A pre-trained FastTuner can significantly enhance optimization with minimal fine-tuning iterations. This adaptability ensures that the model capitalizes on pre-training efforts, minimizing the need for extensive retraining with each new design.
- Decision Transformer Tuner: Our framework leverages a Transformer decoder architecture, offering not only superior decision-making capabilities but also the flexibility for users to

[1]https://github.com/shelljane/REMOTune

fine-tune specific parameter subsets. This architecture captures the intricate interdependencies between parameters, ensuring that tuning is context-aware rather than independent. Additionally, the Transformer tuner allows users to resume tuning from any set of fixed parameters, picking up precisely where it left off. This approach provides a seamless integration interface with other frameworks, facilitating a comprehensive and adaptable tuning process that evolves with the design needs.

- Superior Optimization Performance Across Benchmarks: Our methods consistently outperform academic state-of-the-art (SOTA) approaches by a considerable margin across seven industrial benchmarks and distinct optimization objectives. This consistent performance across various benchmarks underscores the robustness and reliability of our approach, making it a viable solution for industrial applications.

2 MOTIVATIONS AND RELATED WORKS

2.1 Why RL for Parameter Optimization?

Parameter optimization in physical design can be effectively modeled as a sequential decision process, where each timestep involves selecting a parameter value to optimize the final PPA metrics. RL is particularly well-suited for this task due to its ability to learn policies that maximize cumulative rewards over a sequence of decisions. Recently, reinforcement learning (RL) has demonstrated promising success in the realm of electronic design automation (EDA) [19], with notable examples including [10, 16–18, 20–22]. In this work, we propose a fast RL-based learning framework for physical design parameter tuning. Our RL-based approach offers several advantages. Unlike traditional optimization methods, which often suffer from the curse of dimensionality and require complete model retraining when additional parameters are introduced, our approach extends the decision process with additional time steps. This makes it efficient and scalable, even in high-dimensional parameter spaces. Additionally, the continuous learning capability of our model allows it to adapt in real-time with newly collected data. This reduces reliance on static databases and enables immediate updates to the decision policy, resulting in progressively better performance.

By integrating RL with neural networks, such as GNNs, we can effectively capture and leverage the underlying design features. This integration enhances the transferability and generalization of the learned policies in different designs, further improving the overall effectiveness of the tuning process.

2.2 Why Transformer?

We have selected the Transformer [23] as our decision-making agent due to its superior capabilities in sequential tuning. The attention mechanism inherent in the Transformer facilitates sequential decision-making by leveraging previous decisions and automatically discerning and quantifying correlations between parameters across different stages. Moreover, by utilizing the Transformer framework, we extend our tuning approach to parallel a sentence completion task, where the model autonomously continues from the point it left off. In this framework, when provided with a subset of user-specified parameters, our tuner automatically completes the tuning of the remaining parameters to arrive at an optimal solution. This capability allows users to efficiently fine-tune specific subsets of parameters, thereby enhancing the overall effectiveness of the tuning process.

2.3 Why GNN?

State-of-the-art (SOTA) methods have demonstrated strong performance in optimizing physical design parameters; however, their limited transferability remains a significant challenge, as they often require retraining from scratch when applied to new designs. Recently, graph learning has gained

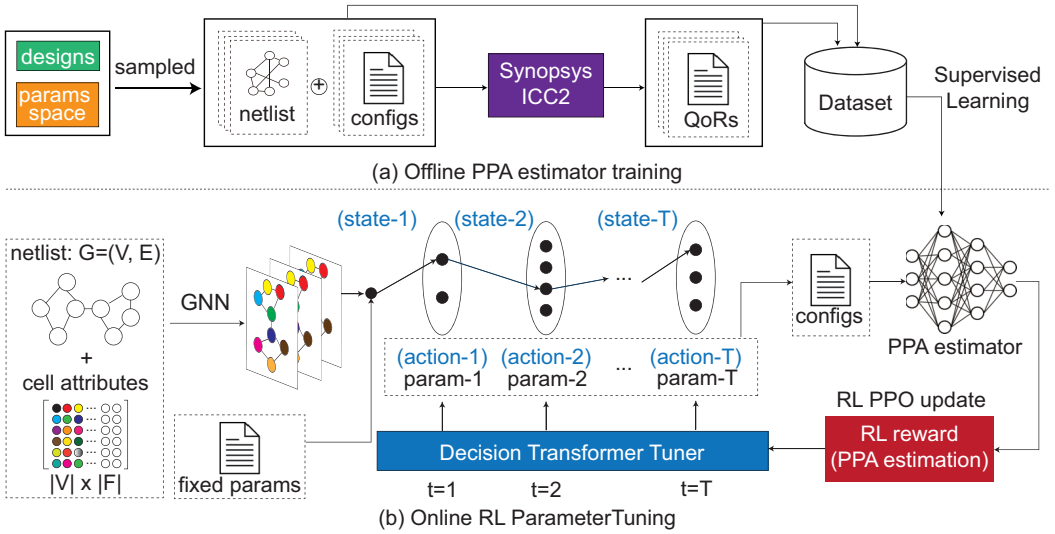

Fig. 1. High-level overview of our hybrid framework: FastTuner optimizes parameters sequentially online by leveraging a pre-trained offline PPA estimator to provide instantaneous reward feedback, eliminating the need for time-consuming ICC2 evaluations.

widespread attention for its ability to embed design netlists into meaningful representations[24]. For instance, the authors of [25] and [26] introduced graph-based learning techniques utilizing Graph Neural Networks (GNNs) to embed netlists effectively, capturing both technology and design-specific parameters. Other notable works, such as [27–35], have further advanced GNN-based approaches in physical design optimization.

Building on this approach, we employ GNNs to utilize knowledge gained from previously optimized designs, enabling our model to generalize and apply tuning experience across a wide range of design scenarios.

2.4 Related Works

In this paper we implement and compare the following ML methods widely adopted for parameter optimization:

Bayesian Optimization (BO): Bayesian Optimization employs a surrogate Gaussian process model to probabilistically predict the PPA based on parameter configurations. With the prediction, an acquisition function guides the selection of the next configuration to sample. The chosen configuration undergoes evaluation with real 3D P&R, and the surrogate model is updated with the new observation. This iterative process enhances the accuracy of the surrogate model over successive steps.

Ant Colony Optimization (ACO): ACO, inspired by real ants, models parameter optimization as a graph. With m parameters, the graph has m+1 nodes, each representing a parameter. Edges between nodes, associated with pheromone levels, denote possible configurations for each parameter. Paths from the first to the last node correspond to parameter configurations. After evaluating performance, pheromone levels on the chosen path are updated based on QoR (quality of result). This iterative process continues until convergence.

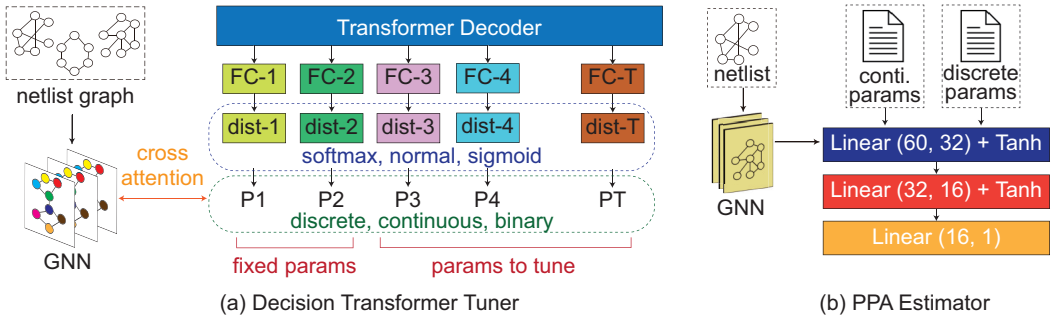

Fig. 2. The detailed architecture of FastTuner comprises the following main components: GNN, Transformer Decoder, Parameter Tuners, and PPA Estimator. The Transformer decoder leverages graph embeddings and predetermined parameter embeddings to sequentially decode parameters using customized parameter predictors.

3 METHODOLOGY

3.1 Overall Optimization Flow

Figure 1 illustrates the overall optimization flow, which integrates both online and offline training strategies. The process begins with an offline phase, where we train PPA estimators using supervised learning on a diverse dataset of parameter-netlist combinations. This dataset is constructed from post-route PPA results derived from various configurations across different designs.

In the online tuning phase, FastTuner leverages netlist embeddings, combined with predefined parameter embeddings as contextual features. At each time step t, the framework uses these embeddings and all previously selected parameters to predict the next action. Once all parameters have been sequentially chosen, they are assessed by the PPA estimator to determine the reward. FastTuner is then updated using the Proximal Policy Optimization (PPO) algorithm based on the calculated reward. This iterative tuning process continues until the system reaches a point where further reward improvements diminish.

3.2 Reinforcement Learning Formulation

Our objective is to sequentially tune a user-selected set of parameters throughout the design flow to achieve optimal post-route PPA. We formalize the problem as a Markov decision process (MDP) and solve it using RL. Our RL formulations are as follows:

- *States* (s): In our setup, a state at time step t, denoted as s_t, consists of two key elements. First, it captures the configuration of the parameters that have been tuned from time steps 1 through $t-1$. These tuned parameters are automatically attended to by the Transformer tuner as hidden features. Second, the state also encompasses the design characteristics of the design currently being optimized. We employ a GNN to encode the essential physical design characteristics.
- *Actions* (a): An action a_t denotes a valid value that can be selected for the parameter being tuned at time step t.
- *Reward* (r): In our framework, the reward is set to zero for each intermediate action $a_1, a_2, \ldots, a_{T-1}$. The final reward is only assigned after the last action, a_T, which receives a reward corresponding to the estimated PPA value derived from the complete P&R process.
- *Trajectory* (τ): A trajectory τ is a complete sequence of parameter selections from time step $t = 1$ to $t = T$, along with the rewards received based on the selected parameters.

3.3 FastTuner Architecture

Our tuning framework consists of five main components, as shown in Figure 2:

(1) PPA estimator: Responsible for providing real-time PPA estimation as a reward.
(2) GNN: This component is responsible for encoding gate-level netlists.
(3) Transformer Decoder: The Transformer decoder sequentially tunes the user-selected parameters
(4) Parameter Tuners: In the final layer, we employ customized modules for each parameter to decode the high-dimensional feature into a probability distribution for action sampling.

FastTuner utilizes GNN to encode the gate-level netlist, generating graph embeddings that capture the structural features of the design. The Transformer decoder in FastTuner begins by processing the predetermined (fixed) parameters. Instead of predicting these parameters, they are directly provided to the decoder using a "teacher forcing" approach, where the predetermined values are input at each step. This approach ensures that the decoder accurately incorporates the fixed parameters as it sequentially predicts the remaining parameters. During this prediction process, the decoder cross-attends with the GNN-derived graph embeddings, allowing the model to integrate detailed design information while making predictions. Cross attention is an essential mechanism to enable the interaction between parameter embeddings, generated by a decoder, and contextual information encoded in graph embeddings. In the proposed decoder-only architecture, parameter embeddings are sequentially generated, while graph embeddings, precomputed using a GNN, serve as static keys and values.

Each parameter embedding, denoted as $\mathbf{p}$, acts as the query ($\mathbf{Q}$), while the graph embeddings, denoted as $\mathbf{g}$, serve as the keys ($\mathbf{K}$) and values ($\mathbf{V}$). The query, key, and value are first linearly transformed into a shared feature space:

$$\mathbf{Q} = \mathbf{W}_q\mathbf{p}, \quad \mathbf{K} = \mathbf{W}_k\mathbf{g}, \quad \mathbf{V} = \mathbf{W}_v\mathbf{g},$$

where $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$ are learned weight matrices.

The attention scores are computed as the scaled dot product of the query and key:

$$\text{Scores} = \frac{\mathbf{Q} \cdot \mathbf{K}^\top}{\sqrt{d_k}},$$

where d_k is the dimensionality of the key, used to stabilize the gradient flow. A softmax function is applied to these scores to normalize them into a probability distribution, highlighting the most relevant components of the graph embeddings:

$$\text{Attention Weights} = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^\top}{\sqrt{d_k}}\right).$$

The attention weights are then used to compute a weighted sum of the value embeddings, extracting relevant contextual information:

$$\text{Output} = \text{Attention Weights} \cdot \mathbf{V}.$$

This output, representing the context derived from the graph embeddings, is integrated back into the parameter embeddings, guiding the decoder to make informed and adaptive decisions.

Each parameter is managed by a customized module tailored to its specific type. While the Transformer layers are shared across all parameters, dedicated prediction layers are employed to generate the final values. In the following subsections, we describe each main component in detail.

Table 1. Initial handcrafted features of each node in our netlist graph.

features	descriptions
wst slack	worst slack of cell
wst output slew	maximum transition of output pin
wst input slew	maximum transition of input pin
drv net power	switching power of driving net
int power	cell internal power
leakage	cell leakage power

3.4 PPA Estimator

The architecture of the provided model comprises a simple feedforward neural network consisting of three linear layers. Each PPA estimator accepts input in the form of concatenated parameter embeddings and graph-extracted features, resulting in a total of 60 dimensions. This input is then processed through two hidden layers with 32 and 16 outputs, respectively, both utilizing Tanh activation functions. Ultimately, the model generates a single output unit for prediction. During training, the model is updated using the mean squared error (MSE) loss, calculated between the ground truth PPA value and the predicted PPA value.

3.5 Netlist Encoding with GNN

To enable our RL agent to generalize across various netlists, we employ GNN to capture both the structural information of the netlists (graph) and the node attributes of the cells (nodes) within the design. Our graph learning process consists of three distinct phases: (1) node-level embeddings (2) graph downsampling and (3) graph-level readout. We initiate the node-level embedding phase with initial node features, which are handcrafted using the metadata associated with each cell, as outlined in Table 1. Subsequently, we iteratively propagate messages from each node to its neighboring nodes. The message-passing mechanism within our GNN is defined as follows:

$$f_{N(u)}^{k-1} = reduce\_mean(\{W_k^{agg} f_v^{k-1}, \forall v \in N(u)\}) \tag{1}$$

$$f_u^k = \sigma(W_k^{proj} \cdot concat[f_u^{k-1}, f_{N(u)}^{k-1}]) \tag{2}$$

In the equations above, we use the symbol σ to represent the activation function, while $N(u)$ refers to the set of neighboring nodes connected to node u. The terms W_k^{agg} and W_k^{proj} represent learnable weights that correspond to the aggregation and projection matrices, respectively. In each iteration, the aggregation function operates on the embeddings of the neighboring nodes in $N(u)$ to produce an aggregated information representation denoted as $f_{N(u)}^{k-1}$. This aggregated message is then combined with the previous embedding of node u, represented as f_u^{k-1}, to update its embedding, which we denote as f_u^k.

The objective of graph downsampling is to create a condensed graph embedding that captures the essential characteristics of the original graph. This process involves utilizing graph attention pooling after each graph convolution layer to selectively retain nodes with the highest attention scores. To derive a holistic representation of the entire graph (netlist), we execute a readout operation (global mean aggregation), to consolidate the node-level embedding obtained from the preceding stages.

Our GNN framework comprises three graph convolution layers followed by a fully-connected (FC) layer, all of which share the same hidden dimension. In our implementation, we set the dimension of the graph convolution layers to 32, and the final FC layer to 16. Consequently, the resulting graph embeddings are represented in a 16-dimensional space.

3.6 Transformer Decoder

Our autoregressive parameter tuning framework offers the distinct advantage of subset tuning, which can be analogized to a next-word prediction task in language modeling. In this approach, when a subset of user-defined parameters is specified, our tuner automatically infers and optimizes the remaining parameters. The self-attention mechanism within the Transformer architecture enables the model to capture the intricate dependencies between the provided subset and the parameters yet to be tuned. We have trained our tuner to adapt to various scenarios. In the first scenario, when only the placement parameters are provided, the tuner predicts and optimizes the clock tree synthesis (CTS) and routing parameters. In the second scenario, when both placement and CTS parameters are provided, the tuner completes the configuration by predicting the routing parameters. Finally, in the absence of any predefined parameters, the tuner autonomously predicts and optimizes the entire set, including placement, CTS, and routing.

3.7 Parameter Tuner

In conventional language model decoders, a shared final softmax layer is typically used to model the distribution of the prediction space, producing final predictions at each time step. However, when the parameters to be predicted vary in their value spaces—such as binary, discrete, and continuous values—relying on a single softmax layer can present substantial challenges. The combination of these diverse parameter types within a single action space results in a high-dimensional and sparsely populated prediction space. This situation demands extensive masking of irrelevant dimensions at each step, which complicates the training process and reduces its overall effectiveness. To address the challenge, we implemented a tailored prediction approach for each parameter within our Transformer tuner. While the Transformer layers are shared, each parameter has its own dedicated prediction layer. This design allows us to handle both discrete and continuous parameters effectively.

For discrete parameters like "`route.global.effort_level`", which can take one of five distinct values ("minimum", "low", "medium", "high", "ultra"), we use a softmax layer with corresponding outputs. For continuous parameters such as "`ccd.max_prepone`", a fully connected layer predicts the mean and standard deviation of a normal distribution.

3.8 RL Update

We update our FastTuner to optimize the expected reward (PPA) by employing the SOTA Proximal Policy Optimization (PPO) algorithm. PPO stands out as a robust RL algorithm that facilitates policy updates while mitigating substantial deviations from the previous policy, which could potentially lead to suboptimal performance or divergence issues. PPO achieves efficient policy optimization by performing multiple update steps for each sample while adhering to a clipped objective function. The PPO clipped surrogate objective that we aim to maximize can be expressed as follows:

$$\mathcal{L}_{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(\rho(\theta)\hat{A},\ clip(\rho(\theta), 1-\epsilon, 1+\epsilon)\hat{A}\right)\right] \tag{3}$$

The objective is to maximize the expected value of the advantage-weighted probability ratio. In other words, PPO encourages actions that have a positive advantage (actions that perform better than expected) and discourages actions that have a negative advantage (actions that perform worse than expected). $\hat{A}$ refers to the advantage function, defined as the difference between the observed rewards R and the expected returns.

$\rho(\theta)$ represents the likelihood of taking a current action under the new policy compared to the old policy, which helps determine how much the policy should be adjusted. The clipped PPO objective introduces a crucial constraint using the min and clip functions to ensure that policy

Algorithm 1: FastTuner training flow.

Input:

$\mathbf{P}_{all} : \{p_1 \ldots, p_n\}$: all parameters
$\mathbf{P}_{fixed} : \{p_1 \ldots p_k\}$: parameters predetermined by the user
$\mathbf{P}_{tuned} : \{p_{k+1} \ldots p_n\}$: parameters to be tuned
GNN for gate-level netlists
Dataset $\mathcal{D} = \{(P_1, G_1, y_1), \ldots (P_N, G_N, y_N)\}$ which contains parameter-netlist combinations and their ground truth PPAs

Output: parameters $\mathbf{P}^*_{tuned} : \{p^*_{k+1} \ldots p^*_n\}$ that optimize the given objectives

(Offline)

Initialize the PPA_Estimator with weights θ_{PPA}
while θ_{PPA} hasn't converge **do**
    sample a batch of data $\mathbf{P}, \mathbf{G}, \mathbf{y}$ from $\mathcal{D}$
    $\mathbf{y}^{'} \leftarrow PPA\_Estimator(\mathbf{P}, \mathbf{G})$
    Compute MSE loss: $J(\theta) \leftarrow \frac{1}{2M}\|\mathbf{y} - \mathbf{y}'\|_2^2$ (M: batch size)
    Update parameters: $\theta_{PPA} \leftarrow \theta_{PPA} - \alpha \cdot \nabla J(\theta)$
end while

(Online)

Initialize the FastTuner with weights θ_{Tuner}
while reward hasn't saturated **do**
    $\mathbf{g} \leftarrow GNN(G)$
    for $i = k + 1 \ldots n$ **do**
        $p'_i \sim FastTuner(p'_i | p'_{k+1}, \ldots, p'_{i-1}; p_1, \ldots, p_k, g)$
        Compute reward: $r \leftarrow PPA\_Estimator(p_1, \ldots, p_k, p'_{k+1}, \ldots, p'_n, g)$
        Update θ_{Tuner} by maximizing Eq. 3 using gradient ascent
    end for
end while

updates are controlled within a specified range, typically denoted as $[1 - \epsilon, 1 + \epsilon]$. ϵ is set to 0.2 by convention. These constraints are vital for preventing excessively large policy updates that could destabilize the training process.

3.9 Training Methodology

Algorithm 1 details the two-phase training process of our FastTuner framework, which consists of both offline and online stages. In the offline phase, referred to as "(Offline)," we begin by initializing a PPA (Power, Performance, Area) estimator, which is subsequently trained through supervised learning. This training utilizes a dataset $\mathcal{D}$, containing various parameter-netlist combinations and their corresponding ground truth PPA metrics. The goal of this phase is to iteratively refine the PPA estimator by minimizing the Mean Squared Error (MSE) loss between predicted and actual PPA values, thus improving its predictive accuracy.

In the online phase, denoted as "(Online)," FastTuner employs a Decision Transformer, which is essentially a Transformer decoder, to sequentially optimize the parameter set. The Transformer tuner incorporates netlist embeddings, generated by a Graph Neural Network (GNN), as contextual information. The tuning process begins either from scratch or after a fixed set of predetermined parameters, if specified. Utilizing the self-attention mechanism, FastTuner sequentially determines the optimal tuning parameters. At each time step, it selects a new parameter as an action based on all previously decided parameters. Once all parameters have been selected, the complete configuration is passed to the PPA estimator, trained during the offline phase, to compute the reinforcement

Table 2. Dataset Statistics: We report two key statistics: the mean (μ) and the coefficient of variation (CV). The CV is defined as the ratio of the standard deviation to the mean (σ/μ).

Design	Power (mW)				WNS (ns)			
	Train		Test		Train		Test	
	Mean	CV	Mean	CV	Mean	CV	Mean	CV
AES	672.310	0.024	671.674	0.023	-0.063	-0.268	-0.058	-0.317
CPU 1	150.510	0.043	148.081	0.048	-0.355	0.195	-0.377	0.204
DMA	149.669	0.029	149.719	0.027	-0.197	-0.101	-0.198	-0.100
ECG	599.524	0.028	588.121	0.018	-0.180	-0.417	-0.118	-0.250
VGA	380.258	0.012	362.305	0.018	-0.503	-0.224	-0.549	-0.272
LDPC	292.468	0.036	292.142	0.035	-0.208	-0.258	-0.172	-0.234

learning (RL) reward. This reward is then used to calculate the Proximal Policy Optimization (PPO) loss, which updates the Transformer Tuner. The process iterates until the reward stabilizes, indicating that the tuning process has converged.

4 EXPERIMENTAL RESULTS

4.1 Experiment Setup

Our tuning framework is developed using Python and the Synopsys IC Compiler II (ICC2) physical design tool. We leverage the deep learning library PyTorch for the overall framework and PyTorch Geometric specifically for graph learning tasks. To evaluate the optimization capabilities of our Transformer Decision Tuner, we conducted a comprehensive comparative experiment against academic state-of-the-art methods, including Ant Colony Optimization (ACO) and Bayesian Optimization (BO). The experiment was designed to explore the design space of selected key parameters in ICC2. Our comparison was carried out across nine benchmarks, all of which are based on the TSMC 28nm technology node.

4.2 Training Dataset

For online learning in reinforcement learning (RL), the model continuously learns from incoming data streams, eliminating the need for a prebuilt dataset. To accelerate RL tuning, we employ supervised learning to train PPA estimators. Our objective is to develop PPA estimators offline that can generalize across a variety of designs, accurately predicting post-route PPA values based on specific parameter configurations. To achieve this, we generated a high-quality, representative dataset by conducting P&R runs using Synopsys ICC2 with different combinations of netlists and parameters. Our data set includes data from over 3500 runs in 7 distinct designs.

The total data collection effort can be estimated based on the runtime of each design, as shown in Table 8. Importantly, this process is a one-time offline effort conducted in parallel across 16 CPU workers. We collected 3 500 total P&R + STA evaluations (500 per design across seven benchmarks). By distributing these evaluations over 16 parallel workers, each design requires only

$$\frac{500}{16} \approx 31.25$$

sequential rounds of evaluation. With an average per-evaluation cost of approximately 2.45 h, this corresponds to roughly 76.6 h (≈3.2 days) of wall-clock time per design. Moreover, the resulting dataset and GNN encoder share experience across all designs, amplifying the benefit of each evaluation and obviating additional P&R + STA runs for subsequent model updates.

By contrast, BO requires 50 sequential runs (≈ 265 h) and ACO requires 55 (≈ 291.5 h) on the largest design, yet still deliver inferior QoR because neither method can leverage cross-design knowledge. Thus, our data-collection overhead is fully amortized ($500/16 < 50, 55$) and in practice

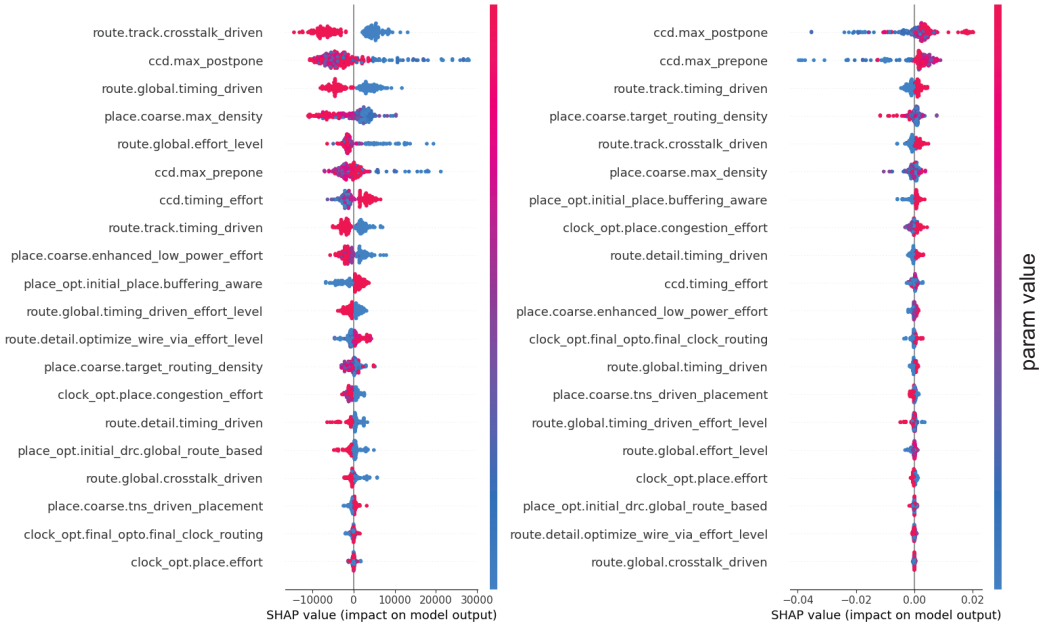

Fig. 3. Parameter analysis using SHAP. (a) Power impact (b) WNS impact. Each dot represents a datapoint for a specific parameter configuration and its PPA metrics. The color gradient (blue to red) indicates parameter values. Dots on the right suggest a positive impact on the model output.

is lower than that of BO or ACO. Finally, the network-training step completes in under one minute, making its runtime negligible compared to data collection.

To visualize our distribution of the dataset, we report two key statistics: the mean (μ) and the coefficient of variation (CV). The coefficient of variation is defined as the ratio of the standard deviation to the mean (σ/μ), as shown in Table 2. As reported, our train-test split is evenly distributed across designs, indicating that there is no significant distributional shift between training and testing data. This balance also serves as evidence of sufficient data collection, which is critical for successful training.

4.3 Parameter Analysis

In our experiment, we systematically explored the design space of ICC2 physical design by identifying 23 critical parameters. These parameters were carefully selected based on domain expertise and statistical analysis, and they encompass both high-level general parameters applicable across the entire design flow and stage-specific parameters relevant to individual stages within the flow. Table 3 provides a comprehensive list of these parameters along with their respective tuning ranges. The parameter set includes both discrete and continuous variables.

To quantify the importance of these parameters, we employed SHAP (SHapley Additive exPlanations), a robust technique for explaining the output of machine learning models. SHAP not only measures the importance of each parameter but also provides interpretable explanations, offering valuable insights into the direction of influence each parameter exerts on the optimization objectives.

Table 3. Parameters we tune and their ranges

Parameter	type	dims or ranges
place.coarse.target_routing_density	float	[0.7, 0.9]
place.coarse.max_density	float	[0.7, 0.9]
place_opt.initial_place.buffering_aware	bool	2
place_opt.initial_drc.global_route_based	int	[0, 1]
placement.aspect_ratio	float	[0.5, 1.5]
ccd.max_prepone	float	[-50 ps, 50 ps]
ccd.max_postpone	float	[-50 ps, 50 ps]
ccd.timing_effort	enum	3
cts.max_skew	float	[0.01, 0.2]
cts.max_fanout	float	[50, 250]
cts.max_buffer_density	float	[0.3, 0.8]
cts.max_latency	float	[0, 1]
route.common.rc_driven_setup_effort_level	enum	4
route.global.effort_level	enum	5
route.global.crosstalk_driven	bool	2
route.global.timing_driven	bool	2
route.global.timing_driven_effort_level	enum	2
route.track.crosstalk_driven	bool	2
route.track.timing_driven	bool	2
route.detail.optimize_wire_via_effort_level	enum	4
route.detail.timing_driven	bool	2
route_opt.flow.enable_power	bool	2
route_opt.flow.enable_irdrivenopt	bool	2

Mathematically, let $f : \mathcal{X} \to \mathbb{R}$ represent a model that maps input instances x from the parameter space $\mathcal{X}$ to predictions. The SHAP value $\phi_i(x)$ for a specific parameter i in a given instance x is defined as:

$$\phi_i(x) = \sum_{S \subseteq \{1,\ldots,p\} \setminus \{i\}} \frac{|S|!(p - |S| - 1)!}{p!} \left[f(x_S \cup \{i\}) - f(x_S) \right] \tag{4}$$

Here, S is a subset of parameters excluding i, x_S represents the instance with only parameters in S, and p is the total number of parameters. This equation computes the marginal contribution of parameter i to the model's prediction by evaluating all possible subsets of parameters. The term $f(x_S \cup \{i\}) - f(x_S)$ captures the change in the model prediction when parameter i is included.

To implement this, we trained an XGBoost model where the input comprises the candidate parameters and the output represents the optimization objectives. After fitting the XGBoost model, we applied SHAP to explain the model and identify the correlation between the parameters and the final objectives. Unlike traditional statistical testing or correlation analysis, SHAP considers all parameters collectively rather than treating them independently. It then marginalizes the impact of each parameter on the model output, providing a more comprehensive and holistic understanding of parameter importance.

The analysis results are depicted in Figure 3. Each dot in the figure represents a datapoint corresponding to a specific parameter configuration and its resulting PPA (Power, Performance, Area) metrics. The color gradient, ranging from blue to red, indicates the value of the parameter, with blue representing lower values and red representing higher values. The position of the dots along the horizontal axis shows the impact of each parameter on the model output—whether it drives the output higher or lower. For instance, if red dots for a particular parameter cluster on the

Table 4. Prediction results of our PPA estimator on the test sets of seven designs: The "CC" refers to the Pearson correlation coefficient.

designs	TNS CC	Power CC	WNS CC
AES	0.97	0.95	0.94
DMA	0.91	0.93	0.90
LDPC	0.95	0.94	0.93
ECG	0.94	0.95	0.90
VGA	0.95	0.92	0.91
Commercial CPU	0.92	0.93	0.90
Rocket	0.90	0.91	0.90

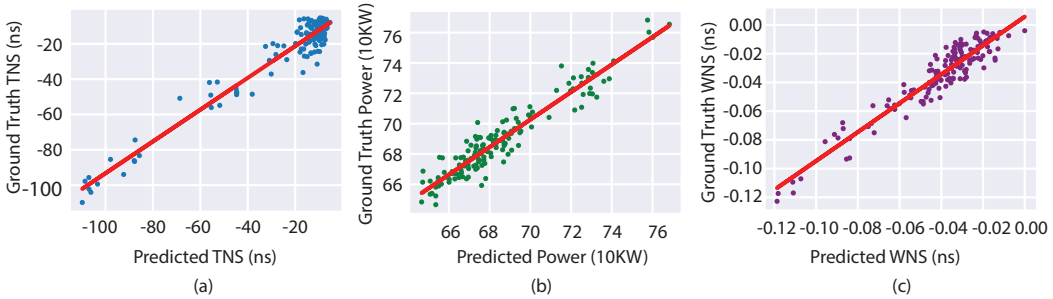

Fig. 4. Correlation analysis of our PPA estimator's predictions on the AES benchmark: For each targeted objective, we plot the scatter distribution between the estimated values (x-axis) and the ground truth values (y-axis). The strong correlation observed in these plots indicates that our estimator effectively provides representative reward estimations..

Table 5. Comparison of FastTuner with ICC2 vs. the PPA Estimator: We trained two versions of FastTuner—one using the PPA estimator and the other using ICC2. We then compared their runtime and PPA results.

	PPA estimator	imp. %	ICC2	imp. %
power (10^5 uW)	5.94	11.48	5.92	11.77
tns (ns)	-28.74	71.62	-28.71	71.64
training iteration	70	-	30	-
time	6 min	-	60 hrs	-

right side of the power metric, this indicates that increasing the value of the parameter tends to significantly increase the total power. Conversely, for metrics like Worst Negative Slack (WNS), a positive impact on timing is observed as the parameter values increase, since a less negative WNS (i.e., closer to zero) signifies improved timing performance.

For instance, as shown in Figure 3, increasing the values of the parameters "route.track.crosstalk_driven," "ccd.max_postpone," and "ccd.max_prepone" demonstrates a marginal contribution to both reducing power consumption (negative power SHAP value) and improving timing (positive WNS SHAP value). Conversely, parameters such as "place_opt.initial_place.buffering_aware" and "ccd.timing_effort" primarily contribute to timing improvement (positive WNS SHAP value) while at the cost of increasing power consumption (positive power SHAP value). Note that the impact is a marginal contribution after considering the interaction of different parameters altogether, which is not intuitively seen by correlation analysis.

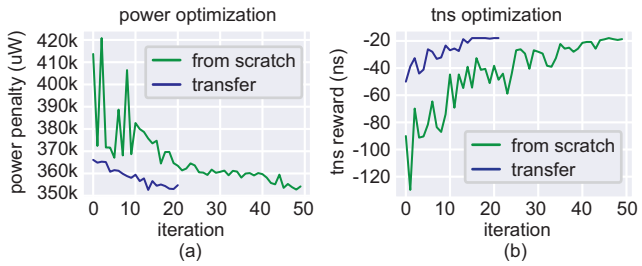

Fig. 5. Transfer learning on VGA using a pre-trained FastTuner model, achieving comparable optimization results at a significantly faster convergence rate. No actual P&R run occurs. Each iteration represents the generation of a configuration for all parameters, evaluation by the PPA estimator, and an update of the tuning agent.

4.4 PPA Estimator Training Results

The PPA estimator takes a pair of netlist and parameter configuration as input and predicts the final PPA value. We utilize supervised learning to train these PPA estimators. During the training process, for each design, we divided the dataset into an 80:20 train-test split, with 80% of the data allocated for training and the remaining 20% reserved for testing. We trained three separate estimators, each dedicated to predicting one of the following metrics: TNS, power, and WNS. For Power-Delay-Product (PDP) estimation, we derived the prediction based on both the power and WNS models. The definition of PDP is as following:

$$\text{PDP} = \text{power} \times (1/\text{target frequency} + \text{WNS}) \tag{5}$$

Figure 4 illustrates the testing results for the AES design across three distinct objectives, showcasing the strong correlations achieved by our model. The data from the AES design was divided into an 80%-20% train-test split, with the testing set comprising unseen data. The figure focuses on the correlation analysis conducted exclusively on the test set, emphasizing the model's ability to generalize and perform accurately on data not used during training. Table 4 presents the Pearson correlation coefficients for various designs and objectives. It's important to emphasize that our primary goal is to estimate rewards for the RL agent, where achieving high correlation is more critical than merely minimizing errors. As shown in both the figure and the table, our estimators consistently exhibit high correlations across different designs. This strong correlation suggests that our estimators effectively distinguish between good and bad outcomes, serving as reliable reward estimations and acting as a suitable proxy for ICC2.

To evaluate the tuning performance using the PPA estimator, we trained two separate FastTuners on the AES benchmark—one using the PPA estimator and the other using real ICC2 feedback, as shown in Table 5. It is important to note that while the PPA estimator may require more training iterations due to the potential instability in reward estimation compared to the ground truth, each iteration is completed in just a matter of seconds. In contrast, obtaining reward feedback from real ICC2 evaluations takes several hours per iteration. Remarkably, we observed that the optimization results from both approaches are nearly identical. As a result, our RL tuning process reduces the design flow parameter tuning to a few minutes, compared to the hundreds of hours it would take to run the place-and-route flow with all these parameters on a given design.

4.5 Transfer Learning Results

The core idea behind transfer learning is to utilize a pre-trained model from one domain to another, enabling zero-shot transfer or faster convergence in unfamiliar domains. In our approach, we

Table 6. Zero-shot FastTuner inference results. We trained FastTuner with four circuits (CPU, AES, DMA, and ECG) first and used it without any further training to solve the three unseen circuits shown in this table. PDP here refers to "power-delay-product."

Metrics	Tool Auto	ACO [36]	BO [3]	FastTuner Zero-shot
LDPC (#cells: 39K, #nets: 41K, #IO: 4.1K)				
Power (10^5 uW)	2.82	2.66	2.58	2.60
TNS (ns)	-150.20	-65.21	-74.77	-66.68
WNS (ns)	-0.22	-0.11	-0.10	-0.12
PDP (10^5W*ns)	2.56	2.35	2.31	2.33
VGA (#cells: 52K, #nets: 52K, #IO: 184)				
Power (10^5 uW)	4.01	3.81	3.68	3.70
TNS (ns)	-88.26	-50.69	-36.54	-46.20
WNS (ns)	-0.38	-0.20	-0.16	-0.20
PDP (10^5W*ns)	2.89	2.68	2.64	2.66
Rocket Core (#cells: 120K, #nets: 120K, #IO: 379)				
Power (mW)	250.80	233.48	229.29	233.40
TNS (ns)	-66.81	-32.45	-21.47	-34.20
WNS (ns)	-0.16	-0.09	-0.07	-0.09
PDP (mW*ns)	140.00	127.28	124.17	127.20

initially employed the same FastTuner model for RL training on specific designs. After completing this training, we loaded the pre-trained FastTuner model's weights on "unseen" designs. The learning curve depicted in Figure 5 illustrates the advantages of transfer learning compared to training from scratch. Through transfer learning, FastTuner rapidly converges to optimization results that are comparable to those achieved by training a new FastTuner from scratch in half the time. This is attributed to the model's ability to generalize using GNN netlist encoding, allowing it to leverage previous training experiences for faster adaptation across various designs.

As shown in Table 6, we trained our FastTuner on four designs: DMA, AES, ECG, and a Commercial CPU, and validated its performance on three distinct, previously unseen designs. Zero-shot FastTuner inference refers to direct inference on unseen designs using the pre-trained FastTuner without any additional training. Remarkably, our FastTuner achieved results comparable to the SOTA even without further training. Additionally, FastTuner can be further fine-tuned for new designs to achieve optimal results, as presented in the next section.

4.6 RL Algorithm Comparison

We experimented with four commonly used RL algorithms to identify the most suitable one for our use case. In the following, we compare the following RL algorithms:

4.6.1 Policy Gradient. Policy Gradient is a simple policy gradient method that directly optimizes parameterized policies. It estimates the gradient of the expected return with respect to the policy parameters and updates the policy in the direction of higher expected returns [37].

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \right] \tag{6}$$

where $\nabla_\theta J(\theta)$ denotes the gradient of the objective function with respect to the parameters θ, $E_{\pi_\theta}[\cdot]$ represents the expectation under the policy π_θ. The term $\nabla_\theta \log \pi_\theta(a|s)$ measures the sensitivity of the policy to changes in θ, and $Q_{\pi_\theta}(s,a)$ is the action value function, indicating the expected return of taking action a in state s under policy π_θ.

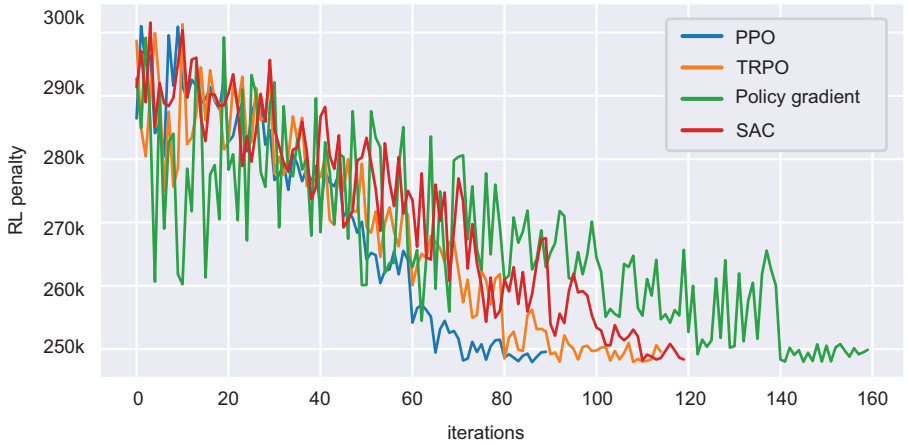

Fig. 6. Performance comparison of RL algorithms in parameter tuning.

4.6.2 Trust Region Policy Optimization (TRPO). TRPO is an iterative approach that optimizes policies while ensuring bounded policy updates. It employs a surrogate objective function constrained by the Kullback-Leibler divergence between consecutive policies, facilitating more stable and monotonic improvements [38].

$$\max_{\theta} \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\theta_{\text{old}}}(s,a)\right] \quad \text{subject to} \quad \mathbb{E}[KL[\pi_{\theta_{\text{old}}}(\cdot|s), \pi_\theta(\cdot|s)]] \leq \delta \tag{7}$$

where the goal is to maximize θ to achieve the maximum expected value of $\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A_{\theta_{\text{old}}}(s,a)$, this expression represents the advantage of action a in state s under the old policy $\pi_{\theta_{\text{old}}}$. The constraint $E[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s), \pi_\theta(\cdot|s)]] \leq \delta$ ensures that the Kullback-Leibler (KL) divergence between the old policy $\pi_{\theta_{\text{old}}}$ and the new policy π_θ does not exceed a threshold δ, thereby controlling the policy update to avoid large deviations from the old policy.

4.6.3 Proximal Policy Optimization (PPO). The core idea of PPO is to restrict the policy update to a trust region [39], similar to TRPO, but using a simpler first-order optimization approach. The PPO objective function is typically formulated as:

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{8}$$

where $L_P^{\text{CLIP}}(\theta)$ is the clipped objective function used in Proximal Policy Optimization (PPO). The expectation $E_t[\cdot]$ is taken in time steps t, and $r_t(\theta)$ is the probability ratio between the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$. The function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ constrains $r_t(\theta)$ to the range $[1-\epsilon, 1+\epsilon]$ to prevent excessively large updates, ensuring stability. The objective function minimizes the expression $\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$, where A_t represents the advantage in the time step t, balancing between maximizing the advantage and maintaining the stability of the policy.

4.6.4 Soft Actor-Critic (SAC). SAC is an off-policy algorithm that maximizes both the expected return and the entropy of the policy. This dual objective promotes exploration and robustness,

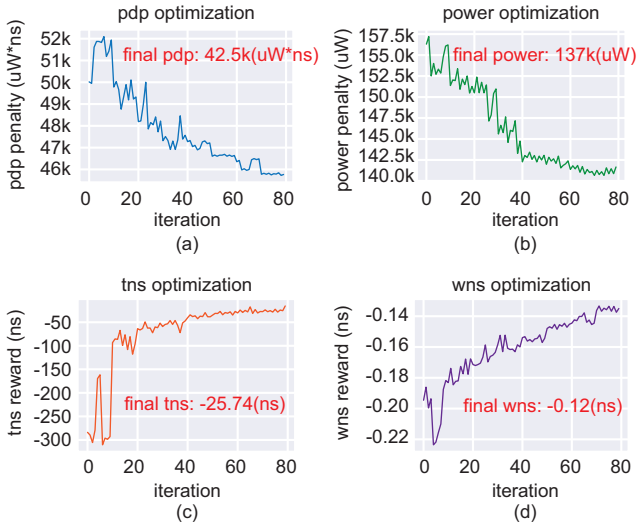

Fig. 7. The learning curves for FastTuner with respect to four different objectives.

which is particularly effective in environments with continuous action spaces [40].

$$J(\pi) = \mathbb{E}\left[\sum_t \gamma^t (r(s_t, a_t) + \alpha H(\pi(\cdot|s_t)))\right] \tag{9}$$

where $J(\pi)$ is the objective function in reinforcement learning, defined as the expected value $E\left[\sum_t \gamma^t \left(r(s_t, a_t) + \alpha H(\pi(\cdot|s_t))\right)\right]$. Here, the summation $\sum_t$ is over all time steps t, γ^t is the discount factor increased to the power t to weigh future rewards less than immediate rewards, $r(s_t, a_t)$ is the reward received at time step t for taking action a_t in state s_t, and $H(\pi(\cdot|s_t))$ represents the entropy of the policy π given the state s_t. The term $\alpha H(\pi(\cdot|s_t))$ is used to encourage exploration by maximizing the entropy of the policy.

The results of the four RL algorithms for power optimization are presented in Figure 6. The Naive Policy Gradient algorithm, while effective, demonstrates rather unstable updates and convergence. This instability is likely due to its lack of mechanisms to constrain updates, making it sensitive to noisy gradients and potentially leading to suboptimal policies. PPO and TRPO stand out as the most robust algorithms in our parameter tuning task. Their robustness can be attributed to the use of a trust region, which constrains policy updates within a specific range, thereby preventing large and potentially harmful updates. This feature is particularly advantageous in uncertain environments, where the reward estimator is noisy compared to the real environment.

The results justify our selection of PPO as the RL algorithm for this task. To further validate the effectiveness of our approach, we evaluated the real outcomes by running the actual P&R process. Since analyzing intermediate performance at every iteration would be overly costly and unnecessary, our analysis focuses on comparing the final QoR after each algorithm has converged. As shown in Figure 6, all four algorithms converged to nearly similar optimal solutions. However, PPO demonstrated the most stable and fastest convergence, outperforming the other algorithms in this case. This confirms PPO's superiority in achieving effective and efficient optimization.

4.7 Overall PPA and Runtime Comparison

In Figure 7, we illustrate the FastTuner learning curves in different optimization objectives using the DMA benchmark. The learning process begins from scratch, taking advantage of the estimated

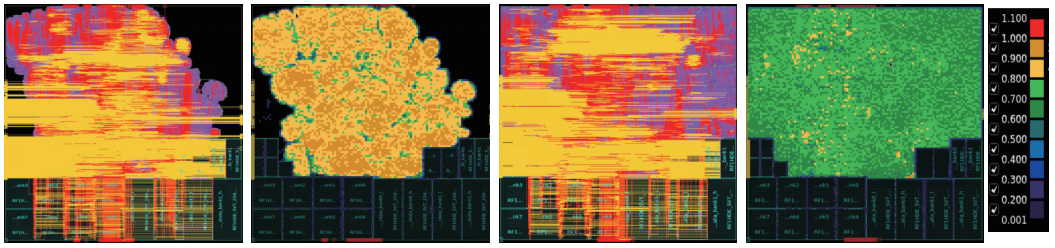

Fig. 8. Layout of "power" optimized design vs. tool default flow design

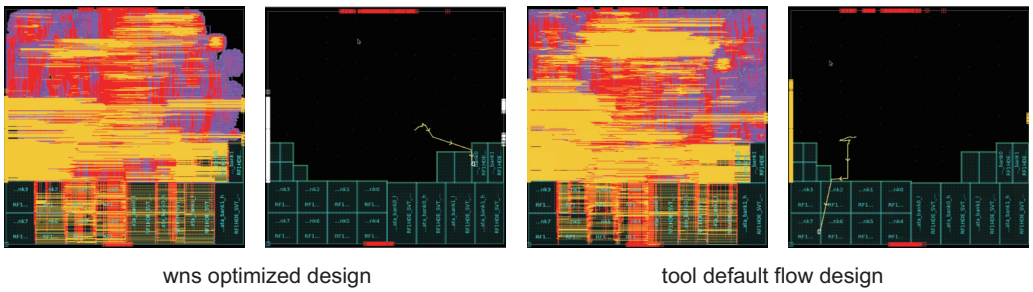

Fig. 9. Layout of "wns" optimized design vs. tool default flow design

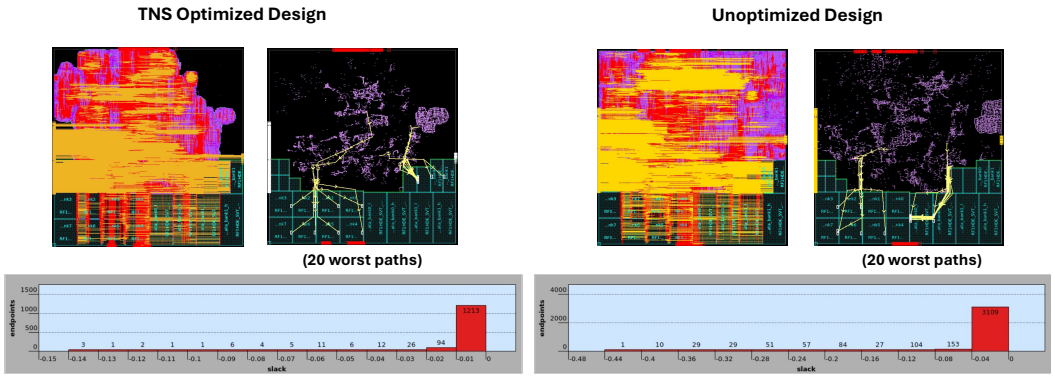

Fig. 10. Layout of "tns" optimized design vs. tool default flow design

rewards generated by our PPA estimators. These curves depict the progression of reward improvements (y-axis) as training iterations advance (x-axis). During each iteration, FastTuner proposes a new parameter setting and undergoes an RL update. The final PPA metrics are validated using Synopsys ICC2. Our observations indicate that FastTuner consistently learns and improves in all objectives. Initially, there are oscillations as it explores the parameter space, but it eventually converges asymptotically towards optimal values. After tuning, the results are evaluated through a full P&R flow using Synopsys ICC2.

Table 8 presents the comparative results of our approach, with the "FastTuner" column reflecting outcomes where all parameters were tuned without any prior fixed settings. Each result was optimized using our PPA estimator and subsequently validated through real Synopsys ICC2 evaluations.

Across all seven benchmarks, our approach consistently outperformed both ACO and BO, achieving significant percentage improvements: up to 79.38% in Total Negative Slack (TNS), 12.22% in total power, and over a 50x reduction in runtime.

A key advantage of FastTuner is its efficiency. Unlike alternative methods that require multiple ICC2 runs, often extending over hours or even days, FastTuner necessitates only a single ICC2 run for final evaluation. Our method delivers near-instantaneous results, with evaluations completed in mere seconds to minutes, thanks to the real-time feedback provided by the PPA estimators. In contrast, BO and ACO struggle with naturally incorporating netlist features into their tuning process, leading to a lack of transferability across different designs.

In Figure 8, we compare our power-optimized design with the default settings of a commercial tool applied to a standard process benchmark design. The analysis reveals that in our power-optimized configuration, standard cell placements exhibit multiple clusters with a high placement density of approximately 80%, in contrast to the tool's default configuration, which shows a lower placement density of around 70%. The more compact placement in our design leads to a routing layout with reduced wire length, as observed from the visibility of metal layers, particularly in the upper right corner where unused routing tracks are evident. This reduction in wire length, in turn, decreases the overall power consumption.

For the second design objective, our framework was configured to optimize the worst negative slack (WNS). Figure 9 provides a comparative overview of the WNS-optimized layout versus the default tool design. A detailed evaluation of the routing and placement layouts for both designs was performed, with the worst timing path highlighted in yellow. In the WNS-optimized design, the critical path is significantly shorter than in the tool's default flow where both paths originate from the register and terminate at the memory macro. Additionally, the routing layout in the WNS-optimized design shows reduced wire length compared to the tool's default configuration, reflecting more efficient standard cell placement and resulting in a shorter timing path with improved WNS.

In the third design objective, the goal was to optimize the total negative slack (TNS) of the design. A comparative analysis between our TNS-optimized layout and the tool's default design is illustrated in Figure 10. The bar chart in the figure shows the distribution of negative timing paths. In the TNS-optimized design, the leftmost bin of the distribution starts at -0.14, indicating an improvement in the left tail compared to the tool's default design, where the leftmost bin starts at -0.44. This improvement in the left tail suggests that the worst timing paths in our optimized design have significantly better slack than those in the default configuration. Additionally, the TNS-optimized design exhibits fewer timing paths with negative slack, further demonstrating that our framework effectively reduces the total negative slack across the design. When comparing the layouts, the TNS-optimized design achieves a more compact placement, similar to the WNS-optimized design but with even greater compactness. This difference arises because TNS optimization focuses on improving the overall slack of all timing paths, whereas WNS optimization targets the single worst-performing path.

4.8 Tuning a Subset of Parameters

Our Transformer framework offers users the flexibility to selectively fine-tune specific subsets of parameters. This capability allows users to retain predetermined parameter subsets that meet their requirements while fine-tuning the remaining parameters. This approach has two key advantages: (1) It eliminates the need to train multiple models for different parameter subsets, and (2) it provides an interface for integration with other tuning methods.

To assess the effectiveness of our selective tuning approach, we trained our FastTuner under two distinct scenarios: (1) Fine-tuning from the CTS stage onwards while keeping default parameters

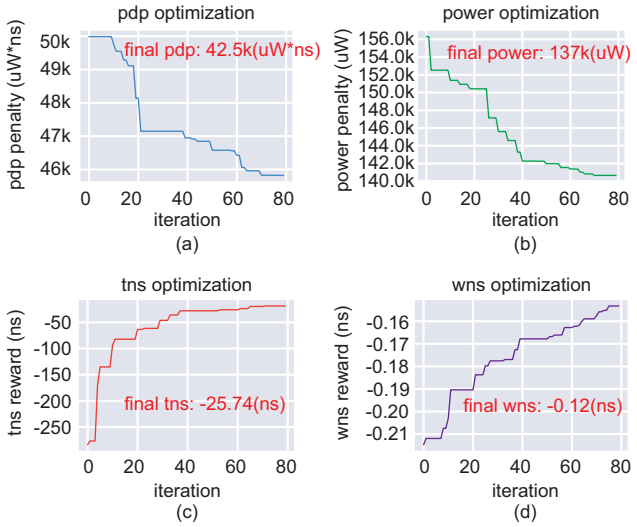

Fig. 11. The learning curves for FastTuner with respect to four different objectives.

for the preceding stages, and (2) fine-tuning from the routing stage onwards while using default parameters for the stages before routing.

The results, presented in Table 8, show that FastTuner delivers competitive outcomes when fine-tuning subsets of parameters. Additionally, we observed that the more parameters we fine-tuned, the better the final results, suggesting that increased parameter tuning flexibility contributes to performance improvements.

5 DISCUSSION

5.1 Why does a sparse reward setting work?

A distinctive feature of parameter optimization with RL is the use of delayed reward settings, where the reward is only observed at the final time step. This arises because, in parameter tuning, defining a meaningful reward without specifying all parameters is inherently challenging. The QoR is often ambiguous when some parameters remain undefined, as different configurations of those parameters can lead to significantly varying outcomes. Arbitrarily shaping intermediate rewards risks misleading the agent and offers no theoretical guarantees of success.

This challenge is common in the literature whenever intermediate rewards cannot be readily defined before all steps have been completed. A widely adopted strategy is to utilize a computationally efficient reward proxy, allowing fast estimation of the final reward and thus enabling more rapid exploration iterations. Several successful RL applications have demonstrated effective learning under similar—or even more demanding—conditions. For example, AlphaGo [41] depends on final game outcomes obtained after hundreds of moves; neural architecture search (NAS) agents [42] receive accuracy measurements only after fully training candidate architectures. These methods commonly handle reward sparsity and delay by employing proxy models or rapid approximations for reward estimation, such as training a reward prediction network or evaluating neural configurations for fewer epochs. In our work, we follow similar practices by introducing a lightweight proxy for rapid, end-of-episode reward evaluation.

Our experimental results (Table 8, Figure 7) demonstrate the effectiveness of our approach. The proposed RL framework consistently identifies parameter sets outperforming state-of-the-art

Table 7. Performance comparison of QoR predictors using Continuous Online Learning and Expert Ensembles, evaluated on DMA, ECG, and Rocket designs. Metrics include the correlation coefficient (CC) and normalized root mean squared error (NRMSE), showing improved accuracy over a single QoR predictor.

	TNS		Power		WNS	
	CC	NRMSE	CC	NRMSE	CC	NRMSE
Original (Single Expert)						
DMA	0.91	0.25	0.93	0.23	0.90	0.24
ECG	0.94	0.20	0.95	0.18	0.90	0.23
Rocket	0.90	0.24	0.91	0.24	0.90	0.26
Continuous Online Learning (Additional Data Points)						
DMA	0.92	0.22	0.93	0.20	0.91	0.18
ECG	0.96	0.16	0.95	0.15	0.93	0.20
Rocket	0.92	0.22	0.92	0.20	0.92	0.20
Expert Ensembles (3 Experts)						
DMA	0.93	0.18	0.94	0.20	0.91	0.20
ECG	0.96	0.12	0.95	0.16	0.92	0.18
Rocket	0.92	0.18	0.92	0.22	0.92	0.22

baselines. Our lightweight PPA estimator enables rapid and cost-effective exploration, significantly reducing overhead.

In Figure 7, by continuously exploring and evaluating new configurations based on its evolving knowledge, the agent avoids premature convergence to suboptimal solutions. Figure 11 illustrates this more clearly by highlighting the best solutions for each objective identified by the agent over successive iterations. This consistent upward trend underscores the agent's capacity to refine its strategy and leverage exploration to overcome the challenges of sparse and delayed rewards, ultimately discovering better solutions as training progresses.

5.2 Dealing with Distributional Shift in PPA estimators

PPA estimators played a crucial role in estimating design performance metrics during optimization. However, distributional shifts can degrade predictor accuracy over time and lead to inaccurate optimization. This subsection explores three approaches to address this challenge.

5.2.1 Continuous Online Learning with Additional Data Points. Continuous online learning involves incrementally updating the PPA estimator dynamically as new data becomes available. By incorporating additional data points generated during the design optimization process, the predictor can adapt to evolving distributions to minimize the risk of Distributional Shift and maintain predictor stability.

5.2.2 Expert Ensembles for Robust Prediction. An ensemble of specialized predictors, or experts, can be employed to handle distributional shifts. Each expert is trained on a subset of the dataset, allowing it to specialize in specific regions. During inference, predictions from the experts are combined using averaging or a voting scheme to ensure robust performance across varying distributions and reduce prediction variance.

5.2.3 Trust Regions for Constrained Prediction. Trust region methods define a localized area in the design space where the PPA estimator is most reliable. By constraining optimization steps to remain within this region, the predictor can maintain higher accuracy, reducing the impact of distributional shift. If exploration outside the trust region is necessary, additional safeguards, such as uncertainty quantification or fallback mechanisms, can guide optimization while minimizing risks.

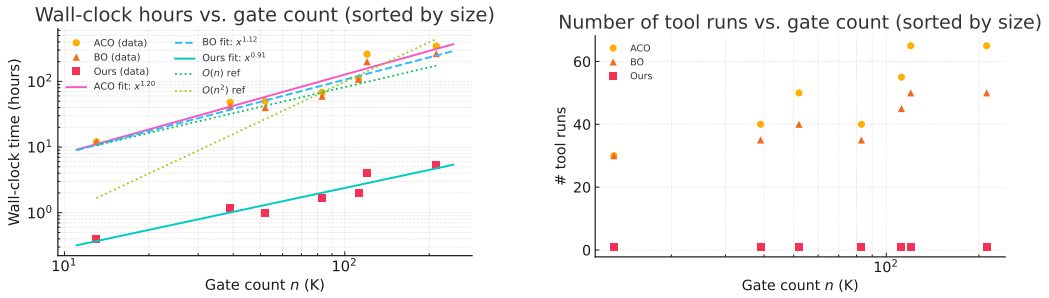

Fig. 12. Scaling with design size. Left: wall-clock hours vs. gate count (log–log) for ACO, BO, and FastTuner; fits use $y = Ax^{\alpha}$ and the legend shows the fitted exponents. Right: number of tool runs vs. gate count. Data are taken from the "runtime (#tool runs / hours)" rows in Table ?? for the seven benchmarks.

Iteratively expanding or refining the trust region as confidence grows helps balance exploration and exploitation.

Table 7 presents the results of experiments conducted using two of the proposed methods: Continuous Online Learning with Additional Data Points and Expert Ensembles.

For the first method, 40 additional data points were collected and appended to the training set for each of the DMA, ECG, and Rocket designs. For the second method, original training set is used with additional data. An ensemble of three experts was trained, with each expert specializing in a different subset of the training dataset. The final prediction was obtained by averaging the outputs of the three experts. Two metrics, the correlation coefficient (CC) and normalized root mean squared error (NRMSE), were used to evaluate performance. As shown in the table, both methods effectively improved prediction accuracy. The correlation coefficient increased, and the NRMSE decreased compared to the results from a single PPA estimator, demonstrating the effectiveness of these approaches in mitigating the effects of distributional shifts.

5.3 How Does GNN help?

GNNs are essential for generalizing learning across different designs. Without GNNs, predictions cannot be effectively shared across data from different designs, hindering knowledge transfer. Additionally, without GNNs, each tuning process must start from scratch, wasting valuable insights and efforts from prior tuning on other designs.

As shown in Table 6, GNNs enable effective transfer learning, significantly accelerating the tuning process for new designs. The results also demonstrate that GNN-based representations support strong zero-shot performance on unseen designs by capturing structural and connectivity patterns that generalize across a family of related circuits. In practice, the model is most effective when the target design shares broad characteristics—such as technology, library, and overall flow setup—with cases it has seen before. Under such conditions, the learned policy adapts readily to moderate variations in netlist structure, macro placement, or block composition, delivering competitive PPA outcomes with minimal additional cost. For designs with characteristics that fall outside this range, the same framework can incorporate additional treatment—such as fine-tuning or re-embedding to capture new structural patterns—so that the learned representation remains aligned with the target design. In contrast, without GNNs, the need to retune from scratch for each design, as reflected in the runtime rows of the aco and bo columns in Table 8, incurs a prohibitively high cost.

5.4 Scalability

Table 8 reports the "runtime (#tool runs / hours)" for seven designs (13K–212K cells). Figure 12 summarizes the trend: on log–log axes, wall-clock time follows a power law with fitted exponents $\alpha \approx 1.20$ (ACO), $\alpha \approx 1.12$ (BO), and $\alpha \approx 0.91$ (our framework). The second panel shows the driver of this gap: ACO/BO require tens of full P&R trials per block (e.g., 65/344.5 h and 50/265 h for the 212K-cell CPU), whereas our framework uses a single ICC2 run for final verification (e.g., 1/5.3 h on the same block), keeping the number of expensive tool invocations constant. This behavior is consistent with the design of our framework: the RL loop uses a pretrained PPA estimator for reward (seconds per iteration) and only performs one ICC2 run at the end. Per our architecture, the netlist is encoded once by a GNN; a forward pass over a netlist graph with n gates and average degree $\bar{d}$ costs $O(L\,|E|\,d) = O(L\,\bar{d}\,n\,d)$ time and $O(n\,d)$ memory for L message-passing layers and hidden width d. The decision model is a Transformer that decodes a short sequence of T parameters; self-attention therefore costs $O(T^2 d)$ per layer independent of n (we do not attend over gates). The PPA estimator is a small MLP on a $\sim$ 60-D feature vector, so its cost is constant in n. Therefore the dominant term in end-to-end runtime is the full P&R call, not the GNN/Transformer/MLP; empirically this is reflected by the constant #runs for our framework and the sublinear exponent we observe in Figure 12. Recently, more and more EDA work has adopted GPU acceleration [43–49, 49–52] to speed up tasks such as parameter optimization, sizing, and placement, which offers another avenue to further alleviate scalability concerns for very large designs.

6 CONCLUSION

We propose the FastTuner framework to address the parameter tuning problem in physical design. Our framework combines online reinforcement learning with offline-trained PPA estimators, eliminating the need for time-consuming P&R processes and reducing tuning times from hours to mere seconds. Additionally, FastTuner incorporates transfer learning via GNNs, enabling it to generalize effectively across diverse design scenarios and achieve academic state-of-the-art performance without requiring complete retraining.

Built on a Transformer architecture, the FastTuner framework enables context-aware tuning, providing users with the flexibility to fine-tune specific parameter subsets and seamlessly integrate with other frameworks. Across seven industrial benchmarks and four distinct optimization objectives, our methods consistently outperform academic state-of-the-art approaches, demonstrating their effectiveness and robustness in various practical applications.

Table 8. PPA and runtime comparison between FastTuner and SOTA [3, 36] methods. TSMC 28nm is used. The 'imp (%)' column indicates the improvement over commercial auto setting in PPA metrics and over SOTA methods in runtime. FastTuner (all) means FastTuner tunes the parameters for all physical design stages, namely, placement, CTS, and routing. FastTuner (CTS+route) means the placement parameters are tuned by ICC2, and CTS and routing parameters by FastTuner. FastTuner uses ICC2 once at the end to collect the final PPA data for verification.

metrics	tool auto	ACO [36]	BO [3]	FastTuner (all)	imp. %	FastTuner (CTS+route)	imp. %	FastTuner (route)	imp. %
Commercial CPU (#cells: 212K, #nets: 216K, #IO: 3.2k)									
power (10^5 uW)	1.54	1.46	1.42	1.39	10.01	1.41	8.44	1.46	5.19
tns (ns)	-70.20	-41.92	-37.66	-18.34	73.87	-26.84	61.77	-45.30	35.47
wns (ns)	-0.22	-0.15	-0.09	-0.08	63.58	-0.09	59.09	-0.17	22.73
pdp (10^5W * ns)	1.73	1.53	1.43	1.34	22.26	1.41	18.50	1.60	7.51
runtime (#tool runs/ hours)	1/5.3	65/344.5	50/265	1/5.3	-	1/5.3	-	1/5.3	-
AES (#cells: 112K, #nets: 112K, #IO: 390)									
power (10^5 uW)	6.71	6.32	6.31	5.94	11.48	6.31	5.96	6.38	4.92
tns (ns)	-101.25	-55.85	-43.56	-28.74	71.62	-35.70	64.74	-56.00	44.69
wns (ns)	-0.08	-0.05	-0.05	-0.03	62.77	-0.05	42.50	-0.06	30.00
pdp (10^5W * ns)	1.51	1.40	1.26	1.20	20.30	1.27	15.89	1.45	3.97
runtime (#tool runs/ hours)	1/2	55/110	45/108	1/2	-	1/2	-	1/2	-
DMA (#cells: 13K, #nets: 14K, #IO: 961)									
power (10^5 uW)	1.52	1.43	1.40	1.37	10.16	1.40	7.89	1.43	5.92
tns (ns)	-96.67	-52.24	-29.13	-25.74	73.38	-40.53	58.07	-56.95	41.09
wns (ns)	-0.21	-0.11	-0.13	-0.12	42.86	-0.16	23.81	-0.17	19.05
pdp (10^5W * ns)	5.08	4.66	4.49	4.25	17.32	4.51	11.22	4.67	8.07
runtime (#tool runs/ hours)	1/0.4	30/12	30/12	1/0.4	-	1/0.4	-	1/0.4	-
ECG (#cells: 83K, #nets: 84K, #IO: 1.7K)									
power (10^5 uW)	6.21	5.83	5.66	5.56	10.49	5.56	10.47	5.86	5.64
tns (ns)	-100.80	-54.37	-41.28	-20.30	79.86	-31.20	69.05	-46.28	54.09
wns (ns)	-0.20	-0.11	-0.12	-0.08	60.35	-0.11	45.00	-0.12	38.00
pdp (10^5W * ns)	2.44	2.25	2.05	1.94	20.40	1.97	19.34	2.26	7.21
runtime (#tool runs/ hours)	1/1.7	40/68	35/59.5	1/1.7	-	1/1.7	-	1/1.7	-
LDPC (#cells: 39K, #nets: 41K, #IO: 4.1K)									
power (10^5 uW)	2.82	2.66	2.58	2.50	11.35	2.65	6.03	2.70	4.26
tns (ns)	-150.20	-65.21	-74.77	-32.52	78.35	-50.20	66.58	-91.20	39.28
wns (ns)	-0.22	-0.11	-0.10	-0.08	64.38	-0.12	47.27	-0.14	37.73
pdp (10^5W * ns)	2.56	2.35	2.31	1.99	22.45	2.12	17.07	2.39	6.64
runtime (#tool runs/ hours)	1/1.2	40/48	35/42	1/1.2	-	1/1.2	-	1/1.2	-
VGA (#cells: 52K, #nets: 52K, #IO: 184)									
power (10^5 uW)	4.01	3.81	3.68	3.52	12.22	3.62	9.73	3.82	4.74
tns (ns)	-88.26	-50.69	-36.54	-18.20	79.38	-29.00	67.14	-58.50	33.72
wns (ns)	-0.38	-0.20	-0.16	-0.15	60.60	-0.18	52.63	-0.21	44.74
pdp (10^5W * ns)	2.89	2.68	2.64	2.27	21.50	2.36	18.34	2.64	8.65
runtime (#tool runs/ hours)	1/1	50/50	40/40	1/1	-	1/1	-	1/1	-
Rocket Core (#cells: 120K, #nets: 120K, #IO: 379)									
power (mW)	250.80	233.48	229.29	228.04	9.07	236.20	5.82	238.19	5.03
tns (ns)	-66.81	-32.45	-21.47	-19.05	71.48	-24.20	63.78	-36.41	45.50
wns (ns)	-0.16	-0.09	-0.07	-0.06	60.45	-0.07	55.63	-0.09	43.21
pdp (mW * ns)	140.00	127.28	124.17	113.02	19.27	114.80	18.00	130.78	6.59
runtime (#tool runs/ hours)	1/4	65/260	50/200	1/4	-	1/4	-	1/4	-

REFERENCES

[1] Piyush Verma. Dso.ai - a distributed system to optimize physical design flows. In Proceedings of the 2024 International Symposium on Physical Design, ISPD '24, page 115–116, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Spectre Simulation Platform and X Spectre. Cadence design systems, inc.

[3] Yuzhe Ma et al. CAD Tool Design Space Exploration via Bayesian Optimization. In 2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD), pages 1–6, 2019.

[4] Hao Geng et al. PTPT: Physical Design Tool Parameter Tuning via Multi-Objective Bayesian Optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 42(1):178–189, 2023.

[5] Hao Geng et al. PPATuner: Pareto-Driven Tool Parameter Auto-Tuning in Physical Design via Gaussian Process Transfer Learning. In Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22, page 1237–1242. Association for Computing Machinery, 2022.

[6] Zheng Zhang et al. A Fast Parameter Tuning Framework via Transfer Learning and Multi-Objective Bayesian Optimization. In Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22, page 133–138. Association for Computing Machinery, 2022.

[7] Su Zheng, Hao Geng, Chen Bai, Bei Yu, and Martin D. F. Wong. Boosting vlsi design flow parameter tuning with random embedding and multi-objective trust-region bayesian optimization. ACM Trans. Des. Autom. Electron. Syst., 28(5), September 2023.

[8] Donger Luo, Qi Sun, Qi Xu, Tinghuan Chen, and Hao Geng. Attention-Based EDA Tool Parameter Explorer: From Hybrid Parameters to Multi-QoR metrics. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6, 2024.

[9] Yi-Chen Lu et al. GAN-CTS: A Generative Adversarial Framework for Clock Tree Prediction and Optimization. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–8, 2019.

[10] Anthony Agnesina et al. VLSI Placement Parameter Optimization using Deep Reinforcement Learning. In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9, 2020.

[11] Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin D. F. Wong. BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration. ACM Trans. Des. Autom. Electron. Syst., 29(1), dec 2023.

[12] Ziyang Yu, Chen Bai, Shoubo Hu, Ran Chen, Taohai He, Mingxuan Yuan, Bei Yu, and Martin Wong. IT-DSE: Invariance Risk Minimized Transfer Microarchitecture Design Space Exploration. In 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pages 1–9, 2023.

[13] Ecenur Ustun et al. LAMDA: Learning-Assisted Multi-stage Autotuning for FPGA Design Closure. In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 74–77, 2019.

[14] Jihye Kwon, Matthew M. Ziegler, and Luca P. Carloni. A Learning-Based Recommender System for Autotuning Design Flows of Industrial High-Performance Processors. In Proceedings of the 56th Annual Design Automation Conference 2019, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Jingyu Pan, Isaac Jacobson, Zheng Zhao, Tung-Chieh Chen, Guanglei Zhou, Chen-Chia Chang, Vineet Rashingkar, and Yiran Chen. Crop: Circuit retrieval and optimization with parameter guidance using llms, 2025.

[16] Hao-Hsiang Hsiao, Yi-Chen Lu, Pruek Vanna-Iampikul, and Sung Kyu Lim. Fasttuner: Transferable physical design parameter optimization using fast reinforcement learning. In Proceedings of the 2024 International Symposium on Physical Design, pages 93–101, 2024.

[17] Hao-Hsiang Hsiao, Sudipto Kundu, Wei Zeng, Wei-Ting J Chan, Deyuan Guo, and Sung Kyu Lim. Insightalign: A transferable physical design recipe recommender based on design insights. In 2025 62nd ACM/IEEE Design Automation Conference (DAC), pages 1–7. IEEE, 2025.

[18] Hao-Hsiang Hsiao, Yi-Chen Lu, Sung Kyu Lim, and Haoxing Ren. Buffalo: Ppa-configurable, llm-based buffer tree generation via group relative policy optimization. In Proceedings of the 44th IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2025.

[19] Ahmet F. Budak, Zixuan Jiang, Keren Zhu, Azalia Mirhoseini, Anna Goldie, and David Z. Pan. Reinforcement learning for electronic design automation: Case studies and perspectives: (invited paper). In 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 500–505, 2022.

[20] Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, and Sung Kyu Lim. Rl-sizer: Vlsi gate sizing for timing optimization using deep reinforcement learning. In 2021 58th ACM/IEEE Design Automation Conference (DAC), pages 733–738, 2021.

[21] Yi-Chen Lu, Wei-Ting Chan, Deyuan Guo, Sudipto Kundu, Vishal Khandelwal, and Sung Kyu Lim. Rl-ccd: Concurrent clock and data optimization using attention-based self-supervised reinforcement learning. In 2023 60th ACM/IEEE Design Automation Conference (DAC), pages 1–6, 2023.

[22] Hao-Hsiang Hsiao, Pruek Vanna-Iampikul, Yi-Chen Lu, and Sung Kyu Lim. Ml-based physical design parameter optimization for 3d ics: From parameter selection to optimization. In Proceedings of the 61st ACM/IEEE Design

Automation Conference, pages 1–6, 2024.
[23] A Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.
[24] Yi-Chen Lu and Sung Kyu Lim. On advancing physical design using graph neural networks (invited paper). In 2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–7, 2022.
[25] Yi-Chen Lu et al. TP-GNN: A Graph Neural Network Framework for Tier Partitioning in Monolithic 3D ICs. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6, 2020.
[26] Yanqing Zhang et al. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6, 2020.
[27] Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, and Sung Kyu Lim. Doomed run prediction in physical design by exploiting sequential flow and graph learning. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9. IEEE, 2021.
[28] Yi-Chen Lu, Siddhartha Nath, Sai Pentapati, and Sung Kyu Lim. Eco-gnn: Signoff power prediction using graph neural networks with subgraph approximation. ACM Trans. Des. Autom. Electron. Syst., 28(4), May 2023.
[29] Yi-Chen Lu, Haoxing Ren, Hao-Hsiang Hsiao, and Sung Kyu Lim. Dream-gan: Advancing dreamplace towards commercial-quality using generative adversarial learning. In Proceedings of the 2023 International Symposium on Physical Design, ISPD '23, page 141–148, New York, NY, USA, 2023. Association for Computing Machinery.
[30] Yi-Chen Lu, Haoxing Ren, Hao-Hsiang Hsiao, and Sung Kyu Lim. GAN-Place: Advancing Open Source Placers to Commercial-quality Using Generative Adversarial Networks and Transfer Learning. ACM Transactions on Design Automation of Electronic Systems, 29(2):1–17, 2024.
[31] Yi-Chen Lu, Siddhartha Nath, Sai Surya Kiran Pentapati, and Sung Kyu Lim. A fast learningdriven signoff power optimization framework. In 2020 IEEE. In ACM International Conference On Computer Aided Design (ICCAD), pages 1–9, 2020.
[32] Yi-Chen Lu, Sai Pentapati, and Sung Kyu Lim. Vlsi placement optimization using graph neural networks.
[33] Jiawei Hu, Pruek Vanna-Iampikul, Zhen Zhuang, Tsung-Yi Ho, and Sung Kyu Lim. Gnn-mls: Signal routing in mixed-node 3d ics through gnn-assisted metal layer sharing. In 2025 62nd ACM/IEEE Design Automation Conference (DAC), pages 1–7. IEEE, 2025.
[34] Zheng Yang, Zhen Zhuang, Bei Yu, Tsung-Yi Ho, Martin D. F. Wong, and Sung Kyu Lim. ML-based fine-grained modeling of DC current crowding in power delivery TSVs for face-to-face 3D ICs. In Proceedings of the ACM International Symposium on Physical Design (ISPD), 2025.
[35] Zheng Yang, Zhen Zhuang, Tsung-Yi Ho, and Sung Kyu Lim. Graph attention-based current crowding analysis at tsv interfaces in 3d power delivery networks. In Proceedings of the 31st Asia and South Pacific Design Automation Conference, 2026.
[36] Rongjian Liang et al. FlowTuner: A Multi-Stage EDA Flow Tuner Exploiting Parameter Knowledge Transfer. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), page 1–9. IEEE Press, 2021.
[37] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, Proceedings of the 31st International Conference on Machine Learning, volume 32 of Proceedings of Machine Learning Research, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.
[38] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.
[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.
[40] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 1861–1870. PMLR, 10–15 Jul 2018.
[41] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. nature, 529(7587):484–489, 2016.
[42] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning, 2017.
[43] Yi-Chen Lu, Zhizheng Guo, Kishor Kunal, Rongjian Liang, and Haoxing Ren. Insta: An ultra-fast, differentiable, statistical static timing analysis engine for industrial physical design applications. In 2025 62nd ACM/IEEE Design Automation Conference (DAC), pages 1–7. IEEE, 2025.
[44] Magi Chen and Ting-Chi Wang. Hyperplace: Harnessing a large language model for efficient hyperparameter optimization in gpu-accelerated vlsi placement. ACM Transactions on Design Automation of Electronic Systems, 2025.

[45] Yi-Hua Chung, Shui Jiang, Wan-Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei Huang. Simpart: A simple yet effective replication-aided partitioning algorithm for logic simulation on gpu. In Proceedings of the European Conference on Parallel Processing, 2025.
[46] Yi-Hua Chung, Nahmsuk Oh, Malleswara Gupta Balabhadra Naga Venkata, Aditya Shiledar, Sudipto Kundu, Vishal Khandelwal, and Tsung-Wei Huang. Accelerating gate sizing using gpu. In Proceedings of the European Conference on Parallel Processing, 2025.
[47] Hao-Hsiang Hsiao, Yi-Chen Lu, Pruek Vanna-Iampikul, Anthony Agnesina, Rongjian Liang, Yuan-Hsiang Lu, Haoxing Ren, and Sung Kyu Lim. Dco-3d: Differentiable congestion optimization in 3d ics. In 2025 62nd ACM/IEEE Design Automation Conference (DAC), pages 1–7. IEEE, 2025.
[48] Yi-Chen Lu, Hao-Hsiang Hsiao, and Haoxing Ren. Llm-enhanced gpu-optimized physical design at scale. 2025.
[49] Yuan-Hsiang Lu, Hao-Hsiang Hsiao, Yi-Chen Lu, Haoxing Ren, and Sung Kyu Lim. Differentiable tier assignment for timing and congestion-aware routing in 3d ics. In Proceedings of the 31st Asia and South Pacific Design Automation Conference, 2026.
[50] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. Pathgen: An efficient parallel critical path generation algorithm. In Proceedings of the 30th Asia and South Pacific Design Automation Conference, pages 416–424, 2025.
[51] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan Luan Lee, Zizheng Guo, Yibo Lin, et al. itap: An incremental task graph partitioner for task-parallel static timing analysis. In Proceedings of the 30th Asia and South Pacific Design Automation Conference, pages 407–415, 2025.
[52] Zhen Zhuang, Zheng Yang, Zhao Yuxuan, Jiawei Hu, Bei Yu, Tsung-Yi Ho, and Sung Kyu Lim. Dpo-3d: Differentiable power delivery network optimization via flexible modeling for routability and ir-drop tradeoff in face-to-face 3d ics. In Proceedings of the 31st Asia and South Pacific Design Automation Conference, 2026.