FastTuner: Transferable Physical Design Parameter Optimization using Fast Reinforcement Learning

Hao-Hsiang Hsiao
thsiao@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Yi-Chen Lu
yclu@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Pruek Vanna-Iampikul
v.pruek@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Sung Kyu Lim
limsk@ece.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

ABSTRACT

Current state-of-the-art Design Space Exploration (DSE) methods in Physical Design (PD), including Bayesian optimization (BO) and Ant Colony Optimization (ACO), mainly rely on black-boxed rather than parametric (e.g., neural networks) approaches to improve end-of-flow Power, Performance, and Area (PPA) metrics, which often fail to generalize across unseen designs as netlist features are not properly leveraged. To overcome this issue, in this paper, we develop a Reinforcement Learning (RL) agent that leverages Graph Neural Networks (GNNs) and Transformers to perform “fast” DSE on unseen designs by sequentially encoding netlist features across different PD stages. Particularly, an attention-based encoder-decoder framework is devised for “conditional” parameter tuning, and a PPA estimator is introduced to predict end-of-flow PPA metrics for RL reward estimation. Extensive studies across 7 industrial designs under the TSMC 28nm technology node demonstrate that the proposed framework FastTuner, significantly outperforms existing state-of-the-art DSE techniques in both optimization quality and runtime. where we observe improvements up to 79.38% in Total Negative Slack (TNS), 12.22% in total power, and 50x in runtime.

CCS CONCEPTS

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

KEYWORDS

Physical Design, Reinforcement Learning

ACM Reference Format:
Hao-Hsiang Hsiao, Yi-Chen Lu, Pruek Vanna-Iampikul, and Sung Kyu Lim. 2024. . In *Proceedings of the 2024 International Symposium on Physical Design (ISPD ’24), March 12–15, 2024, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626184.3633328

1 INTRODUCTION

Modern commercial Physical Design (PD) tools offer a wide range of tunable parameters to configure underlying sophisticated optimization engines subject to various design requirements. However, with the increasing design complexity driven by Moore’s Law and the stringent time-to-market productization demand, traditional PD parameter optimization methods, including the “auto-tuning” approaches in most commercial tools, have become impractical due to their significant runtime that takes from days to weeks. Therefore, an efficient and generalizable Design Space Exploration (DSE) technique is urgently needed to find the optimal parameter configuration that results in best-in-class Power, Performance, and Area (PPA) metrics based on design features.

Numerous automated DSE techniques have arisen to tackle the challenges posed by extensive PD design spaces. Two notable examples are [1], which leverages a generative adversarial network (GAN) to enhance clock tree prediction, and [2], which introduces a deep reinforcement learning approach for optimizing placement parameters. However, both of these approaches are constrained to addressing only specific stages within the PD workflow and require a significant level of expertise to configure a near-optimal solution.

To address these limitations, there is a growing interest in developing alternative automated approaches for optimizing the entire set of PD parameters. For instance, in [3], an ACO-based evolutionary algorithm was employed to iteratively fine-tune parameters in groups. Other for example [4] proposed a learning-assisted autotuning framework that utilizes the XGBoost algorithm and design-specific features extracted from the early stages of the FPGA design flow to achieve design closure. Nevertheless, these heuristic approaches are time-consuming and susceptible to getting trapped in local optima.

On the other hand, recent state-of-the-art (SOTA) methods utilize Bayesian-based techniques to optimize power, performance, and area (PPA) objectives, for example, [5], [6], [7], and [8]. While all these methods have shown their effectiveness, there is potential for improvement. For example, these approaches still necessitate a full P&R run to evaluate a single set of parameter combinations. Moreover, in addition to capturing the correlations between parameters and PPA metrics, it’s important to harness the underlying design features to accelerate the optimization of future designs.

In this paper, we present a fast RL online tuning approach based on offline-trained PPA estimators that provide instantaneous rewards rather than time-consuming P&R feedback. Our RL tuner utilizes GNN to capture the underlying design features, enabling generalization to unseen designs. The FastTuner is built upon a Transformer encoder-decoder architecture, providing users with the flexibility to fine-tune specific parameter subsets and seamlessly integrate with existing tuning methods to achieve superior performance. Our main contributions are as follows:

- We propose a hybrid FastTuner framework that enables online RL tuning using offline-trained PPA estimators, eliminating the need for the time-consuming P&R process, thus reducing the tuning process from hours to seconds.
- FastTuner facilitates transfer learning through GNN, enabling it to generalize across various design scenarios. A pretrained Fast-Tuner can significantly improve optimization results with just a few iterations of fine-tuning.
- Our framework utilizes an encoder-decoder Transformer architecture, offering users the flexibility to selectively fine-tune specific subsets of parameters and providing an interface for seamless integration with other frameworks.
- Our methods consistently demonstrated superior results compared to SOTA approaches by a significant margin across 7 industrial benchmarks and distinct optimization objectives.

2 MOTIVATIONS

2.1 Why RL for Parameter Optimization?

RL excels in learning a policy that maximizes long-term rewards through a sequence of decisions. In contrast to other SOTA algorithms, such as Bayesian optimization (BO), which suffers from the curse of dimensionality as the number of parameters increases, RL offers distinct advantages when dealing with a growing number of parameters. We frame the problem of parameter tuning as a sequential decision-making task. As the number of parameters increases, traditional methods often encounter a dimensionality problem. However, in our RL-based approach, this expansion in the parameter count merely adds a few decision time steps, which is typical in RL tasks where tens to hundreds of such steps are common, making it effective for handling high-dimensional parameter spaces.

As an online tuning algorithm, our tuner enables continuous learning with newly collected data points, reducing the reliance on an extensive database or requiring only a relatively small one. Additionally, our RL method can be seamlessly integrated with neural networks to leverage design features and enhance transferability.

Although RL has demonstrated success in various domains, one of the challenges in applying RL to physical design lies in the slow turnaround time (TAT) of the reward signal, often requiring a full P&R process. Taking inspiration from [9], we have created a fast and transferable PPA estimator using GNN to provide instant reward estimations, which allows us to accelerate RL updates by performing planning on the PPA estimator, thereby circumventing the time-consuming P&R process.

2.2 Why Transformer? Why GNN?

We have chosen the Transformer language model as our decision-making agent as it stands out as the optimal choice for sequential tuning. Firstly, our parameter optimization process involves a sequence of decisions, making a language model a natural fit since it has the unique capability to capture critical information from previous observations and actions, which is vital in our context.

Secondly, optimizing across multiple design stages requires careful consideration of the intricate interdependencies among these stages. Identifying these dependencies can be challenging, as they vary in strength and direction, ranging from positive correlations to orthogonality. Fortunately, the Transformer architecture possesses a self-attention mechanism, which automatically discerns and quantifies correlations between parameters.

Lastly, by leveraging the Transformer encoder-decoder framework, we extend our tuning approach to resemble a sentence completion task. In this setup, when provided with a subset of user-specified parameters, our tuners autonomously complete the tuning of the remaining parameter set with the optimal solution. This capability empowers users to efficiently fine-tune any selected subset of parameters, enhancing the overall effectiveness of our approach.

The authors of the paper [10] present an unsupervised graph-based learning framework for tier partitioning. This innovative approach allows the algorithm to effectively understand technology and design-related parameters. As discussed in the previous section, while SOTA methods have demonstrated competitiveness in optimizing physical design parameters, they often lack transferability, necessitating retraining from scratch when encountering a new design. To overcome this limitation, we harness GNN to leverage knowledge gained from prior optimized designs. This empowers our model to generalize across various design scenarios.

3 FASTTUNER METHODOLOGY

The goal of FastTuner is to automatically tune the parameters of any physical stage in the shortest possible time, aiming to achieve the best post-route PPA.

Figure 1 provides a high-level overview of our hybrid FastTuner framework. In the offline training phase, PPA estimators are trained using supervised learning on diverse parameter-netlist combinations. During online tuning, FastTuner utilizes netlist embedding, incorporating predetermined parameter embeddings as contextual features. At each time step t, FastTuner takes into account the contextual feature and all past actions (from a_1 to a_{t-1}) attended by the self-attention mechanism to sample a new action a_t. Once all parameters are sampled, they are forwarded to the PPA estimator for reward estimation. FastTuner is updated using the RL algorithm PPO, and the online tuning iterates until reward saturation. The newly sampled parameter-netlist combinations are stored in the sample buffer for offline PPA estimator improvement.

3.1 Reinforcement Learning Formulation

Our objective is to sequentially tune user-selected set of parameters throughout the workflow to achieve optimal post-route PPA. This problem falls under the category of combinatorial optimization problems, and we have formalized it as a Markov decision process (MDP) and solved it using RL. we formally describe our RL formulations:

- *States* (s): In our context, a state at time step t, denoted as s_t, includes two key components. First, it captures the configuration of the parameters that have been tuned from time steps 1 through $t-1$. These tuned parameters are automatically considered by the self-attention model as hidden input features. Second, the state also encompasses the specific design that is currently under optimization. We employ 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.

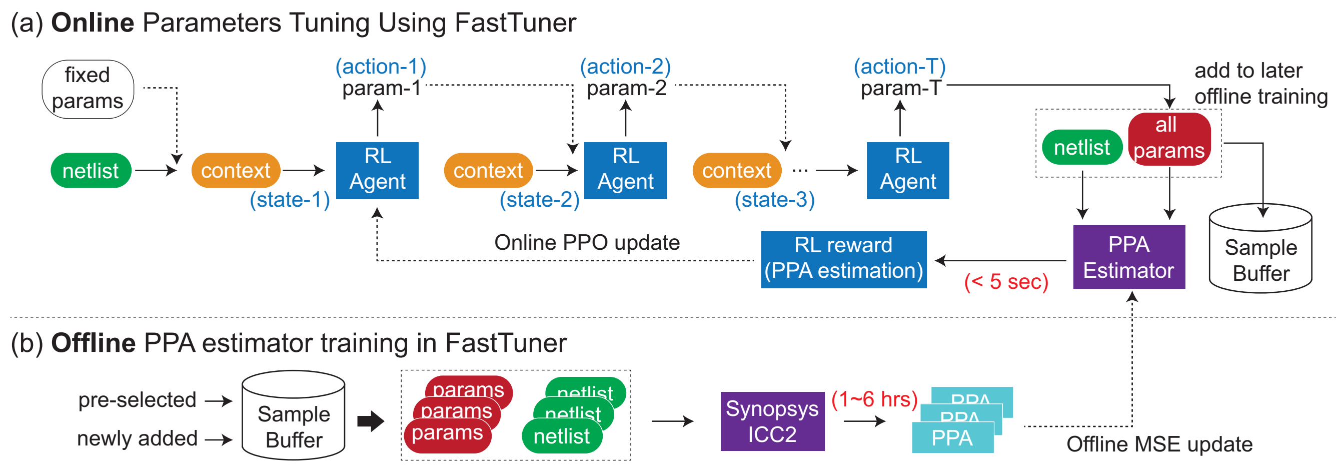

Figure 1: High-level overview of our hybrid framework. FastTuner optimizes parameters sequentially online by leveraging the pre-trained offline PPA estimator to provide reward feedback, preventing the need for time-consuming ICC2 feedback.

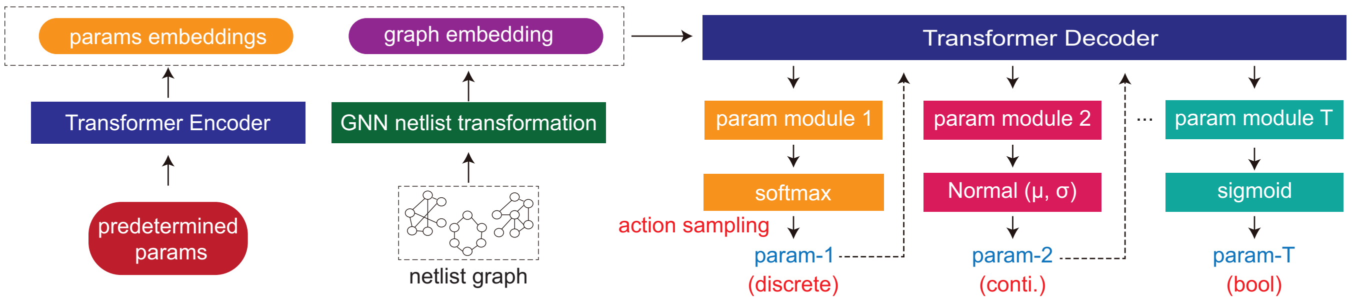

Figure 2: The detailed architecture of our FastTuner includes the following main components: GNN, Transformer Encoder, Transformer Decoder, and Parameter Tuners. The Transformer decoder utilizes graph and predetermined parameter embeddings to sequentially decode parameters using customized parameter predictors.

- *Reward* (r): In our setup, rewards are set to zero for intermediate actions $a_1, a_2, \ldots, a_{T-1}$, with the exception of the last action, a_T, which corresponds to the estimated PPA value obtained after the entire P&R process.
- *Trajectory* (τ): A trajectory τ encompasses the complete sequence of parameter selections from time step $t = 1$ to $t = T$, along with the corresponding rewards received.

3.2 Overall Architecture

Figure 1 provides a high-level overview of our FastTuner framework. We propose a hybrid tuning framework that combines both online and offline techniques. We first train our PPA estimators offline on different parameter-netlist combinations using supervised learning. Specifically, we construct a training dataset comprising post-route PPA results obtained from various parameter combinations across different designs, utilizing ICC2.

In online tuning phase, FastTuner utilizes netlist embedding, incorporating predetermined parameter embeddings as contextual features. At each time step t, FastTuner takes into account the contextual feature and all past actions $(a_1, \ldots, a_{t-1})$ attended by the self-attention mechanism to sample a new action a_t. Once all parameters have been sequentially sampled, they are forwarded to the PPA estimator for reward estimation. Subsequently, FastTuner is updated using the RL algorithm PPO based on the obtained reward. This online tuning process continues until the reward reaches a saturation point. During each online tuning iteration, the newly sampled parameter-netlist combination is appended to the sample buffer for later offline improvement of the PPA estimators.

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 Encoder: The Transformer encoder encodes the user-predetermined parameters.
(4) Transformer Decoder: The Transformer decoder sequentially tunes the user-selected parameters
(5) 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 and employs a Transformer encoder for predetermined parameters. The Transformer decoder processes contextual features, including graph embeddings and predetermined parameter embeddings, to sequentially decode the remaining parameters. Each parameter is associated with a customized module tailored to its type. Transformer layers are shared across all parameters, while dedicated prediction layers for each parameter generate predicted values.

Below, we describe each main component in detail:

3.3 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.

3.4 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:

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

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

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.

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

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.

3.5 Transformer Encoder and Decoder

Instead of conventional parameter tuning techniques involving the adjustment of a predefined set of parameters, we have adopted a more flexible approach inspired by sentence completion tasks. In this setup, when provided with a subset of user-defined parameters, our tuning mechanism autonomously takes charge of optimizing the remaining parameter set to achieve the best solution.

The predefined parameters are used as input context and undergo embedding by the Transformer encoder. The encoder leverages the Transformer's attention mechanism to grasp contextual meaning and generate contextual features through self-attention. In contrast to a directional Long Short-Term Memory (LSTM) network, the Transformer encoder processes the entire information sequence simultaneously. This capability allows the model to capture the full context of the predefined parameters. Following this, the contextual features are passed to the Transformer decoder, which guides the finalization of the tuning process based on the provided context.

By utilizing both the graph features and predetermined parameters as contextual information, the Transformer decoder performs autoregressive tuning of the parameters. In each time step, the decoder's self-attention mechanism not only considers contextual features but also naturally attends to all previously predicted parameters, ranging from p_1 to p_{t-1}. Consequently, it captures the intricate interdependencies among parameters, illustrating how the earlier stages of physical design influence the tuning of subsequent parameters. The decision-making process at time step t can be expressed as follows: $p_t \sim \pi_\theta(p_t | p_1, .., p_{t-1}, s_t)$.

3.6 Parameter Tuner

In a typical language model decoder, a final softmax layer is employed to model the distribution of the prediction space and generate final predictions for each time step. However, when the parameters to be predicted at each time step belong to different value spaces, employing a single softmax layer can introduce challenges. Mixing binary, discrete, and continuous parameters within a unified action space can lead to a high-dimensional, sparse space that is challenging to train effectively.

To overcome this challenge, we have devised a tailored prediction approach for each parameter within our Transformer tuner. While the Transformer layers are shared across all parameters, each parameter possesses its own dedicated prediction layer responsible for generating its predicted values. This design affords us the flexibility to predict discrete and continuous parameters with ease.

For discrete parameters such as "route.global.effort_level," which can take on five distinct values ("minimum", "low", "medium", "high", and "ultra"), we employ a softmax layer with five outputs to predict the probability of each value. Conversely, for continuous parameters such as "ccd.max_prepone," we employ a fully-connected layer that predicts both the mean and standard deviation of a normal distribution.

3.7 Training Methodology

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 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.

Algorithm 1 illustrates the offline and online training process of our FastTuner framework. In the offline phase, denoted as "(Offline)," we initialize a PPA estimator and train it using supervised learning with a dataset $\mathcal{D}$ containing diverse parameter-netlist combinations and their corresponding ground truth PPA metrics. This phase iteratively improves the PPA estimator by updating its parameters based on the Mean Squared Error (MSE) loss between predicted and actual PPA values.

In the online phase, denoted as "(Online)," FastTuner leverages the trained PPA estimator and GNN for netlist embeddings. It employs a Transformer-based architecture with an encoder-decoder structure. FastTuner utilizes both predetermined parameter embeddings and netlist embeddings as context and applies the self-attention mechanism to make sequential decisions for tuning parameters. At each time step, it samples a new parameter action. Once all parameters are sampled, it calculates a reward using the PPA estimator. The algorithm updates the FastTuner's parameters using the PPO algorithm until the reward stabilizes. Additionally,

Algorithm 1: FastTuner training flow.

```
Input:
 1: $\mathbf{P}_{all} : \{p_1 \ldots, p_n\}$: all parameters
 2: $\mathbf{P}_{fixed} : \{p_1 \ldots p_k\}$: parameters predetermined by the user
 3: $\mathbf{P}_{tuned} : \{p_{k+1} \ldots p_n\}$: parameters to be tuned
 4: GNN for gate-level netlists
 5: 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)
 1: Initialize the PPA_Estimator with weights $\theta_{PPA}$
 2: while $\theta_{PPA}$ hasn't converge do
 3:     sample a batch of data $\mathbf{P}, \mathbf{G}, \mathbf{y}$ from $\mathcal{D}$
 4:     $\mathbf{y}' \leftarrow PPA\_Estimator(\mathbf{P}, \mathbf{G})$
 5:     Compute MSE loss: $J(\theta) \leftarrow \frac{1}{2M}\|\mathbf{y} - \mathbf{y}'\|_2^2$ (M: batch size)
 6:     Update parameters: $\theta_{PPA} \leftarrow \theta_{PPA} - \alpha \cdot \nabla J(\theta)$
 7: end while
(Online)
 8: Initialize the FastTuner with weights $\theta_{Tuner}$
 9: while reward hasn't saturated do
10:     $\mathbf{g} \leftarrow GNN(G)$
11:     $\mathbf{m} \leftarrow FastTuner\_Encoder(p_1, \ldots, p_k)$
12:     for $i = k+1 \ldots n$ do
13:         $p'_i \sim FastTuner\_Decoder(p'_i | p'_{k+1}, \ldots, p'_{i-1}, m, g)$
14:         Compute reward: $r \leftarrow PPA\_Estimator(p_1, \ldots, p_k,$
            $p'_{k+1}, \ldots, p'_n, g)$
15:         Update $\theta_{Tuner}$ by maximizing Eq. 3 using gradient ascent
16:     end for
17: end while
```

the newly sampled parameter-netlist combinations are stored for offline improvement of the PPA estimator in future iterations.

4 EXPERIMENTAL RESULTS

4.1 Experiment Setup

Our framework is implemented in Python, leveraging deep learning libraries PyTorch and PyTorch Geometric. To assess our results, we conduct a comparative analysis against SOTA methods, namely ACO and BO. This evaluation involves exploring the parameter space of the physical design tool across seven industrial benchmarks. All these benchmarks are designed based on TSMC 28nm technology nodes, and the physical design process is executed using Synopsys IC Compiler II (ICC2).

For parameter optimization, we select over 20 tool parameters, as presented in Table 2, and tune them on various benchmarks to optimize diverse objectives including power, TNS, WNS, and PDP. These parameters exhibit different data types, encompassing discrete, floating-point, and integer values.

4.2 PPA Estimator Training Results

Te expedite RL tuning, we explore supervised learning to train PPA estimators. Our goal is to train PPA estimators capable of generalizing across various designs, predicting post-route PPA values based on specific parameter configurations. To accomplish this, it is crucial to establish a representative dataset. To construct this, we conducted numerous P&R runs, each involving varying pairs of netlists and parameters with randomly generated combinations.

Table 2: Parameters we tune and their ranges, which includes all PD stage's parameter from placement, CTS, and routing.

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

Table 3: Prediction results of our PPA estimator on the validation sets of seven designs. "CC" denotes 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

In total, our dataset comprises data collected from more than 3500 runs spanning seven distinct designs.

During the training process, for each design, we divided the dataset into an 80:20 train-test split, where 80% of the data was allocated for training, and the remaining 20% was reserved for testing. We trained three separate estimators, each focused on predicting TNS, power, and WNS. In the case of PDP estimation, we based it on both the power and WNS models. Figure 3 displays the validation results for the AES design across three distinct objectives, highlighting the high correlation our model has achieved across these objectives. Table 3 presents the Pearson correlation coefficients for various designs and different objectives. It's important to note that our primary goal is to estimate rewards for the RL agent, and as such, we prioritize achieving high correlation over merely minimizing errors. Our achievement of this high correlation indicates that our estimators provide effective reward estimations, distinguishing between good and bad outcomes.

To assess tuning performance with 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 4. It's worth noting that, while the PPA estimator may demand more training iterations, each iteration is completed within a matter of seconds. In contrast, obtaining reward feedback from real ICC2 evaluations takes several hours per iteration. Remarkably, we found that the optimization results from both approaches are nearly identical. Consequently, our RL tuning process has been dramatically reduced from hundreds of hours to just a few minutes.

Table 4: FastTuner with ICC2 vs. the PPA estimator. We trained two FastTuners: one with the PPA estimator and one with ICC2. We compare 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	-

4.3 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 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 4 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 5, 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.

4.4 Overall PPA and Runtime Comparison

In Figure 5, we present the learning curve of our FastTuner for various optimization objectives using the DMA benchmark. The learning process starts from scratch and is based on the estimated rewards provided by our PPA estimators. These curves show the progression of reward improvement estimated by the PPA estimator (y-axis) as learning proceeds (x-axis). In each training iteration, FastTuner proposes a parameter setting and undergoes an RL update. The final PPA results are evaluated using ICC2. We observe that FastTuner effectively learns across all objectives, initially exploring the parameter space with oscillations, and subsequently converging nearly asymptotically toward optimal values. After tuning, we evaluate the result through a full P&R using Synopsys ICC2.

The results are presented in Table 6. The "FastTuner" column represents tuning all parameters without specifying fixed parameters beforehand. All results are optimized using the PPA estimator

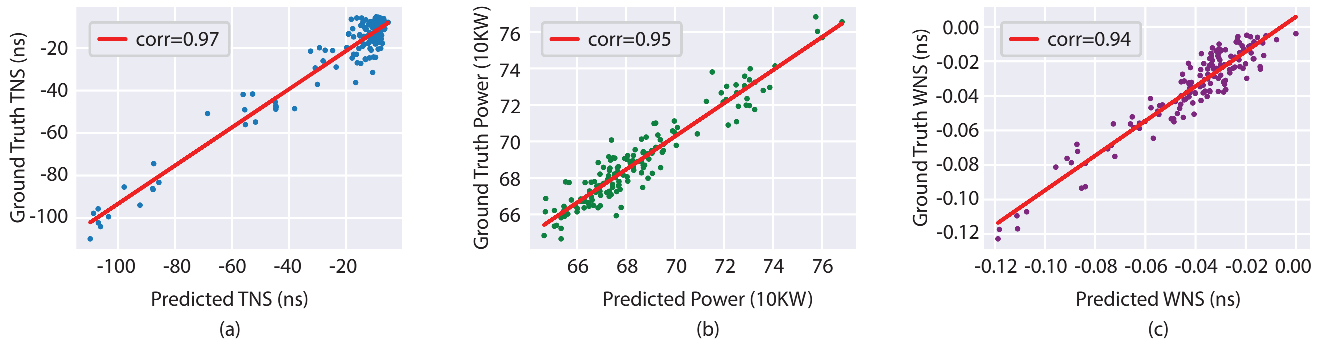

Figure 3: Correlation analysis of our PPA estimator's predictions on the AES benchmark. For each targeted objective, we plot the scatter distribution between the estimated values (x-axis) and the ground truth values (y-axis). The high correlation indicates that our estimator is capable of providing a representative reward estimation.

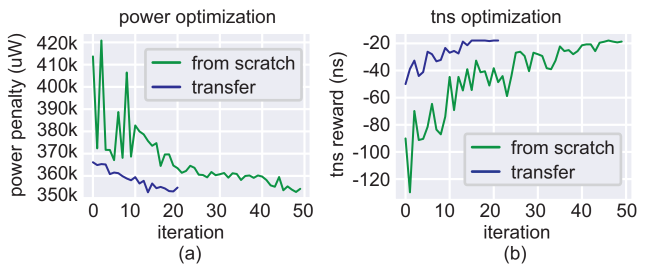

Figure 4: Transfer learning on VGA using a pre-trained FastTuner model, achieving comparable optimization results at a significantly faster convergence rate.

Table 5: 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.

metrics	tool auto	aco [3]	bo [5]	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

and assessed through real Synopsys ICC2 evaluations. The results demonstrate that our approach consistently outperforms ACO and BO across all seven benchmarks, showcasing a significant improvement percentage. Notably, our method requires just a single ICC2 run for result evaluation, whereas other methods necessitate numerous tool runs that can span from hours to days. In contrast, our approach is nearly instantaneous, taking only seconds to minutes, thanks to the real-time rewards provided by PPA estimators.

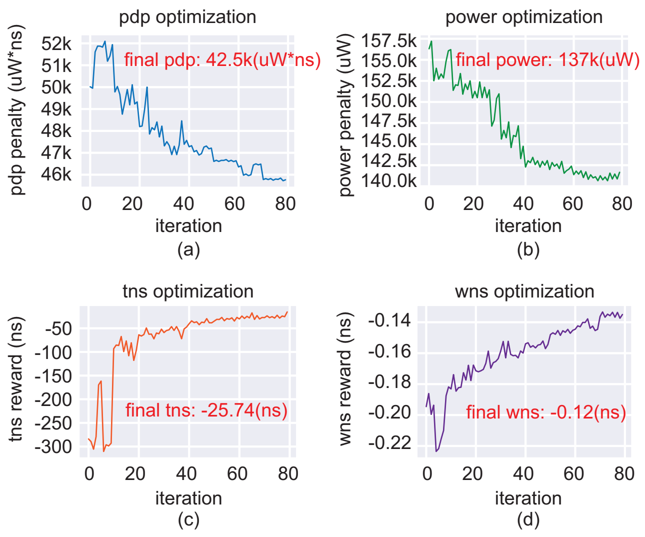

Figure 5: The learning curves for FastTuner with respect to four different objectives.

4.5 Tuning a Subset of Parameters

Our encoder-decoder framework provides users with the flexibility to selectively fine-tune specific parameter subsets. This enables users to retain predetermined parameter subsets they find satisfactory and fine-tune the remainder. The advantages are twofold. Firstly, there is no need to train multiple models for different parameter subsets. Secondly, it offers a versatile interface for integration with other tuning methods. To evaluate the performance of our selective tuning approach, we trained our FastTuner under two distinct scenarios: (1) Fine-tuning from the CTS stage onwards while using default parameters for those preceding CTS. (2) Fine-tuning from the route stage onwards while using default parameters for those preceding route.

Table 6: PPA and runtime comparison between FastTuner and SOTA [3, 5] 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 [3]	bo [5]	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%	1.40	9.09%	1.47	4.85%
tns (ns)	-70.20	-41.92	-37.66	-18.34	73.87%	-26.85	61.75%	-45.29	35.49%
wns (ns)	-0.22	-0.15	-0.09	-0.08	63.58%	-0.09	58.04%	-0.16	25.63%
pdp (10^5uW * ns)	1.73	1.53	1.43	1.34	22.26%	1.40	19.08%	1.59	8.11%
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%	6.29	6.23%	6.39	4.82%
tns (ns)	-101.25	-55.85	-43.56	-28.74	71.62%	-35.64	64.80%	-58.61	42.12%
wns (ns)	-0.08	-0.05	-0.05	-0.03	62.77%	-0.04	44.85%	-0.05	31.94%
pdp (10^5uW * ns)	1.51	1.40	1.26	1.20	20.30%	1.26	16.56%	1.43	5.08%
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%	1.39	8.36%	1.41	7.25%
tns (ns)	-96.67	-52.24	-29.13	-25.74	73.38%	-40.52	58.08%	-56.93	41.11%
wns (ns)	-0.21	-0.11	-0.13	-0.12	42.86%	-0.14	33.33%	-0.15	28.57%
pdp (10^5uW * ns)	5.08	4.66	4.49	4.25	17.32%	4.50	11.39%	4.68	7.88%
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%	5.58	10.09%	5.85	5.75%
tns (ns)	-100.80	-54.37	-41.28	-20.30	79.86%	-30.98	69.27%	-46.26	54.11%
wns (ns)	-0.20	-0.11	-0.12	-0.08	60.35%	-0.10	50.00%	-0.12	38.36%
pdp (10^5uW * ns)	2.44	2.25	2.05	1.94	20.40%	1.97	19.42%	2.26	7.23%
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%	2.64	6.21%	2.69	4.64%
tns (ns)	-150.20	-65.21	-74.77	-32.52	78.35%	-48.21	67.90%	-91.48	39.10%
wns (ns)	-0.22	-0.11	-0.10	-0.08	64.38%	-0.12	47.29%	-0.14	36.98%
pdp (10^5uW * ns)	2.56	2.35	2.31	1.99	22.45%	2.12	17.05%	2.38	6.84%
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%	3.60	10.22%	3.81	4.92%
tns (ns)	-88.26	-50.69	-36.54	-18.20	79.38%	-28.35	67.88%	-59.06	33.08%
wns (ns)	-0.38	-0.20	-0.16	-0.15	60.60%	-0.17	54.42%	-0.20	47.04%
pdp (10^5uW * ns)	2.89	2.68	2.64	2.27	21.50%	2.35	18.69%	2.64	8.70%
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%	235.22	6.21%	238.19	5.03%
tns (ns)	-66.81	-32.45	-21.47	-19.05	71.48%	-24.15	63.85%	-36.41	45.50%
wns (ns)	-0.16	-0.09	-0.07	-0.06	60.45%	-0.07	58.10%	-0.09	43.21%
pdp (mW * ns)	140.00	127.28	124.17	113.02	19.27%	115.60	17.43%	130.78	6.59%
runtime (#tool runs/ hours)	1/4	65/260	50/200	1/4	-	1/4	-	1/4	-

The results are presented in Table 6. We observed that our FastTuner delivers competitive results when fine-tuning subsets of parameters. Furthermore, we noted a phenomenon where the more parameters we were able to tune, the better the final results, suggesting that a greater degree of parameter tuning flexibility leads to improvement.

5 CONCLUSION

In summary, we introduce the FastTuner framework, which significantly reduces tuning times from hours to seconds by utilizing RL with PPA estimators. FastTuner also leverages transfer learning via GNN, enabling effective generalization across various designs. Across multiple industrial benchmarks and optimization objectives, FastTuner consistently outperforms SOTA by a substantial margin, demonstrating its effectiveness and robustness.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant No. CNS 2137288 and the industry members of the Center for Advanced Electronics in Machine Learning.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.