RL-Sizer: VLSI Gate Sizing for Timing Optimization using Deep Reinforcement Learning

Yi-Chen Lu[12], Siddhartha Nath[2], Vishal Khandelwal[3], and Sung Kyu Lim[1]
[1]School of ECE, Georgia Institute of Technology, Atlanta, GA
[2]Synopsys Inc., Mountain View, CA; [3]Synopsys Inc., Hillsboro, OR
yclu@gatech.edu; siddhartha.nath@synopsys.com; vishal.khandelwal@synopsys.com; limsk@ece.gatech.edu

***Abstract*—Gate sizing for timing optimization is performed extensively throughout electronic design automation (EDA) flows. However, increasing design sizes and time-to-market pressure force EDA tools to maintain pseudo-linear complexity, thereby limiting the global exploration done by the underlying sizing algorithms. Furthermore, high-performance low-power designs are pushing the envelope on power, performance and area (PPA), creating a need for last mile PPA closure using more powerful algorithms. Reinforcement learning (RL) is a disruptive paradigm that achieves high-quality optimization results beyond traditional algorithms. In this paper, we formulate gate sizing as an RL process, and propose RL-Sizer, an autonomous gate sizing agent, which performs timing optimization in an unsupervised manner. In the experiments, we demonstrate that RL-Sizer can improve the native sizing algorithms of an industry-leading EDA tool, *Synopsys IC-Compiler II* (ICC2), on 6 commercial designs in advanced process nodes ($5-16nm$). RL-Sizer delivers significantly better total negative slack (TNS) and number of violating endpoints (NVEs) on 4 designs with negligible power overhead, while achieving parity on the others.**

I. Introduction

Gate sizing for power, performance, and area (PPA) optimization is the backbone of modern physical design (PD) flows, which is used extensively from synthesis to signoff. It is an algorithmic process of assigning an appropriate size (gate type) to each optimizable design instance from a set of equivalent standard cell libraries under different process, voltage, and temperature (PVT) corners. For an instance, the number of available gate sizes is discrete and is limited by the underlying technology. This makes gate sizing an NP-hard problem [8], where the solution space scales exponentially with respect to the size of netlist.

Existing gate sizing algorithms in electronic design automation (EDA) tools are based on various pseudo-linear heuristics or analytical methods driven by (statistical) static timing analysis (STA) that easily result in globally sub-optimal sizing solutions. As the benefit of technology scaling saturates, leading edge high-performance low-power design flows are seeking to make the final PPA boost by leveraging more powerful sizing algorithms, even at the cost of runtime (or, increased turn-around time (TAT)), in order to achieve the desired PPA scalability at advanced process nodes. Therefore, time-to-market and best-in-class PPA requirements create a push-pull situation in EDA flows under advanced technologies (e.g., $16nm$ to $5nm$).

Reinforcement learning (RL) is a promising machine learning (ML) paradigm that has been demonstrated to achieve super-human performance in many high-dimensional control problems [12]. In the realm of EDA, a recent work [7] shows that how RL may be used for macro placement to improve design TAT and PPA. In addition, RL is applied to solve transistor sizing for analog designs [13], global routing [3], and technology mapping [9]. Nonetheless, we have to invent and engineer a different RL algorithmic framework to solve our gate sizing problem due to the significantly larger solution space compared with these previous works.

The goal of this work is to build the first high-dimensional RL framework, RL-Sizer, which formulates the classic gate sizing problem as an RL process and solves it by applying advanced RL algorithms equipped with graph neural networks (GNNs). To demonstrate the feasibility of the proposed RL formulation, we specifically focus on the problem of gate sizing for timing optimization at the post-route stage, where the goal is to optimize the total negative slack (TNS) of a design. Unlike prior works [1, 11] that perform aggressive optimization based on meta-heuristics or non-generalizable analytical methods that assume convexity of the objective functions, our RL agent optimizes design performance in a more global and flexible (i.e., customized loss function) manner with the consideration of design and technology features (multi-corner multi-mode) encoded by GNNs.

The outcome of our effort is a universal RL-based gate sizing framework that performs timing optimization across various advanced technologies for industrial-scale designs. To our knowledge, this is the first work that formulates the classic gate-sizing problem as an RL problem (control problem), and presents advanced RL algorithms equipped with graph representation learning techniques to solve it. The contributions of this work are as follows:

- We present RL-Sizer, the first-ever RL-based gate sizing algorithm for timing optimization. RL-Sizer achieves competitive TNS optimization results to an industry-leading commercial tool on six commercial designs using advanced technology nodes ($5nm, 12nm, 16nm$), and specifically, in four designs, RL-Sizer can significantly outperform the commercial sizing engine on TNS and number of violating endpoints (NVEs) with negligible total power overhead.
- We develop a graph-based feature encoder using GNNs

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

that captures the instance characteristics related to timing optimization. These encoded features are taken as the inputs of RL-Sizer and are proven to be highly useful.

- We demonstrate the effectiveness of our "local-graph" method for fast timing approximation. For a target instance, we elegantly take the TNS change of its "local three-hop neighborhood graph structure" (termed as local-graph) as the reward of the sizing move taken instead of the entire netlist. This local-graph approximation can be easily threaded across various instances, which significantly accelerates the learning process.

II. Reinforcement Learning Formulation

A. Gate Sizing as a Control Problem

The gate sizing problem can be intuitively formulated as a Markov Decision Process (MDP), as there are many sequential decisions made iteratively regarding sizes of gates on critical (and, sometimes sub-critical) paths to achieve target timing closure. Therefore, we can conceptually apply RL algorithms to solve it (i.e., maximize the reward of this process). Given a set of design instances to be sized for timing optimization, we train an RL agent to sequentially determine their final gate sizes. Here we present key terminologies and concepts of an RL process, and illustrate how they are mapped to the gate sizing problem in our work.

- *State (s)*: A state s represents a "design instance", which is realized by concatenating the encoded features of its local three-hop neighborhood (by GNNs), and the technology features extracted from libraries.
- *Action (a)*: An action a refers to the "new gate size" assigned to the design instance in state s. In the implementation, it is realized as the "driving strength change" Δd. Assume an instance whose current size has strength d. After taking an action (a sizing move), it is assigned the gate size in the technology whose strength is the closest to $d' = d + \Delta d$ among all possible choices.
- *Reward (r)*: A reward r is the outcome of performing an action a on an instance in state s. In our case, it represents the TNS change of the instance's "local-graph". For each sizing iteration, the goal of RL-Sizer is to maximize the *total reward* (sum of individual rewards) of all instances.
- *Trajectory (τ)*: A trajectory τ refers to a sizing iteration (an RL process), from time step $t = 0$ to $t = T$ (final time step). At each time step t, there is a corresponding state s_t, action a_t, and reward r_t pair denoted as (s_t, a_t, r_t). Note that a complete gate sizing run in commercial tools consists of multiple trajectories.

Figure 1 shows an illustration of our RL gate sizing process, where we consider each selected instance as a unique *RL state* and determine their new gate sizes sequentially (the instance selection algorithm is illustrated in Section III). Note that STA update using a commercial tool is performed once at the end of a sizing iteration, which provides the *RL reward* for each action taken. We want to emphasize that instances in a common sizing iteration (*RL trajectory*) are not independent of each other. Actions (gate sizes) that are taken in previous time steps (prior instances) will contribute to the sizing decision of the current time step. We leverage a policy gradient algorithm named Deep Deterministic Policy Gradient (DDPG) [4] to capture this dependency and to optimize the total reward.

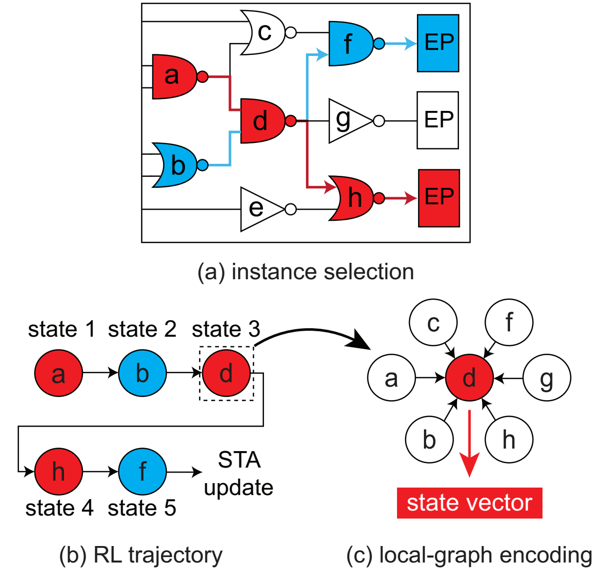

Fig. 1: Illustration of our RL gate sizing process. (a) Input netlist with 3 end points (EPs). First, we identify the worst critical path in the design (red), and then for each endpoint, we identify the most negative slack path (e.g. blue) overlapping with the design critical path. Finally, instances on these paths (the design critical path and the other paths overlapped with it) are selected for one sizing iteration. (b) Sort the selected instances in topological order, and determine their final gate sizes sequentially by considering each of them as an *RL state*. STA is performed after all selected instances are assigned new sizes. (c) Example of local-graph encoding using GNNs on gate "d". The encoded state vector is taken as the input of the RL agent to determine the action (new gate type).

B. Our Key Concept: Local-Graph Approximation

As mentioned earlier, we propose the concept of "local-graph" for *RL state* encoding (Figure 1(b)) and *RL reward* approximation. Given a target instance, the "local-graph" of this instance refers to its local "three-hop neighborhood graph structure" from the netlist. The rationale is two-fold. First, the final gate size of a target instance not only depends on the characteristics of itself, but also the behavior of its neighbors (e.g. the capacitive load that this target instance is driving). We leverage GNNs (to be elaborated in Section III-C) to encode such neighboring information into a vector as an *RL state* vector, which serves as the input of the RL agent for the decision of the corresponding *RL action* (i.e., new gate size).

Second, the timing impact of a gate sizing move on a design instance to the overall netlist diminishes as the hop count increases. Therefore, instead of taking the total design TNS change as the *RL reward* of a sizing move (ideal case, but computationally expensive), we take the TNS change of its local-graph. This way, the reward gives fast and good fidelity approximation, while offering an opportunity for parallel com-

putation. That is, an improvement in local-graph TNS mostly results in positive design (netlist) TNS change, and vice versa.

C. Graph Representation Learning

GNNs have revolutionized many research areas [2] by performing effective graph representation learning that encodes graph information into meaningful embeddings through a message passing scheme. Since VLSI netlists are represented as hypergraphs, we can apply GNNs to them. Recently, many studies have demonstrated the great potency of applying GNNs to solve EDA problems, such as transistor sizing [13], layout decomposition [10], power estimation [5, 14], and circuit partitioning [6]. We leverage GNNs to distill netlist features that are related to timing. Specifically, given a target instance for sizing, we utilize GNNs to encode the features within its local-graph (3-hop neighborhood), and take the encoded features as the input of the RL framework to determine its final gate size.

III. RL-Sizer Algorithms

In this work, we focus our problem on the post-route stage, which is the PD stage that designers struggle the most for timing optimization. However, our method generalizes to other stages of the PD flow as well. The goal of our framework, RL-Sizer, is to optimize the design performance in terms of TNS by making good sizing moves on combinational instances. Note that we do not size sequential instances.

A. Overview

Figure 2 shows a high-level overview of our framework. First, we develop an instance selection algorithm to select the combinational instances that must be sized to improve design TNS. These selected instances form a sizing iteration (an *RL trajectory*). Note that a complete gate sizing run consists of multiple sizing iterations. For each selected instance, we use GNNs to encode its "local-graph" (described in Section II), and take the encoded features along with the technology features that represent the driving strength, capacitance, and slew constraints as the *RL state* s_t. We define the corresponding *RL reward* r_t subject to the *RL action* a_t taken at time step t as the TNS change on its local-graph. Note that as aforementioned, each selected instance belongs to a unique time step and is sized sequentially from time step $t = 0$ to $t = T$ (last instance). This order is based on netlist topology.

At each time step t, the objective of RL-Sizer is to maximize the long-term return G_t, which is denoted as

$$\max_{\theta} G_t(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{k=0}^{T} \gamma^k r_{t+k} \right], \tag{1}$$

where π denotes the policy function (network) parameterized by θ, which takes the state s_t as input and outputs the action a_t, γ denotes the reward discount factor. To maximize this objective G, we perform gradient descent on the policy parameters θ using the DDPG [4] loss function update. In the following sub-sections, we present each component of our framework in detail.

Algorithm 1 Instance selection for a sizing iteration (*RL trajectory*).

Input: $G = (V, E)$: a post-route netlist.
Output: $V' \in V$: selected instances to be sized.

1: Run full-chip STA.
2: $W \leftarrow$ current worst negative slack (WNS) path in the design
3: Initialize $V' \leftarrow$ {non-overlapping instances in W}
4: $\{P\} \leftarrow$ for each endpoint, identify its worst negative path
5: **for** $p \in \{P\}$ **do**
6:     **if** p is overlapping with W **then**
7:         **for** $v \in p$ **do**
8:             **if** v's local-graph does not overlap with $\{V'\}$'s **then**
9:                 add instance v on path p to set V'
10: $V' \leftarrow$ topological_sort(V')   ▷ linear time, achieved by DFS

TABLE I: Initial node features for GNN encoding.

features	descriptions
slack	worst slack of paths through instance
in_slew	worst input pin slew
out_slew	output pin slew
arc_delay	worst cell arc (input to output pin) delay
nom_delay	nominal delay (fan-out of 4)
cell_cap	cell capacitance
drv_length	driving (output) net length
drv_load	sum of driving capacitance (net + cell)
drv_res	sum of driving resistance
fanin_cap	average capacitance of fan-ins
sibling_cap	sum of capacitance of siblings

B. Instance Selection

Selecting feasible instances that can possibly improve design TNS is essential to the success of RL-Sizer. Algorithm 1 presents our instance selection process. Given a routed design $G = (V, E)$, where V denotes the design instances and E denotes the connections, our algorithm identifies the target instances $V' \in V$ that will further be sized sequentially by RL-Sizer in (netlist) topological order.

Note that as shown in the algorithm, selected instances in V' do not share "overlapping" local-graphs, which means instances in a sizing iteration do not overlap in their local 3-hop neighborhood. This is to minimize the sizing impact between each other, since in our settings (as shown in Figure 2), instances in a common iteration are sized simultaneously (i.e., an STA update is performed once per iteration). Ideally, one can perform an STA update per sizing move of an instance to completely address the issue of interference. However, this approach is impractical due to the computation expense of STA on VLSI designs. In our experiments, we find that with the proposed technique of "non-overlapping local-graphs", RL-Sizer can effectively determine the feasible size for each selected instance that optimizes TNS.

C. Encoding RL State using GNNs

1) Initial Node Features: Prior to the local-graph encoding using GNNs, we compute the initial node-specific features for each design instance as shown in Table I. These features are carefully chosen based on domain expertise and are expected to characterize an instance's sizing impact to design timing. However, these features are not sufficient for RL-Sizer to determine the gate sizes that optimize design performance,

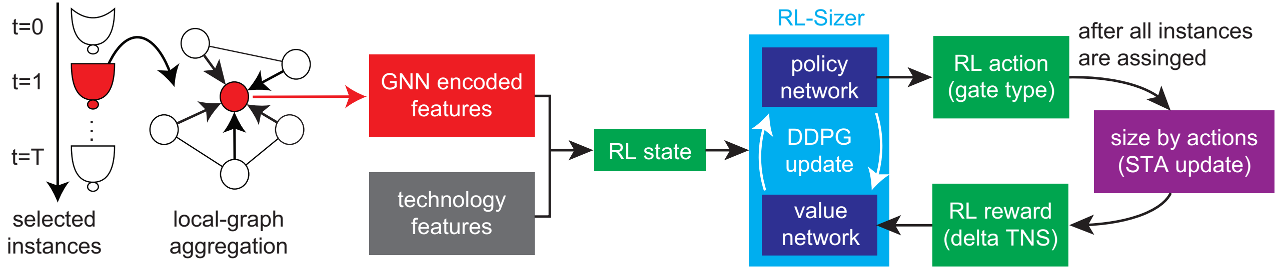

Fig. 2: Overview of our RL-Sizer framework. Given the selected instances from Algorithm 1, for each instance (e.g. red), we take its encoded local-graph features along with the technology information as the *RL state* s_t, and leverage RL-Sizer to determine the *RL action* a_t assigned. An STA update is performed when all selected instances are assigned new gate sizes. Finally, we take the "local-graph TNS change" as the *RL reward* of each action taken. Rewards across time steps (instances) are leveraged to update RL-Sizer through the DDPG algorithm [4].

because the final gate size of a target instance not only depends on these features, but also the information from its neighboring nodes. Therefore, we use GNNs as a local-graph encoder to obtain better representations in graph-level. Note that the initial features are not normalized instance-wise, since for each sizing iteration, we select a new set of instances to be sized.

2) Local-Graph Encoding: Based on the initial node features defined in Table I, we leverage GraphSAGE [2], a variant of GNNs, to encode local-graph features for each selected instance. Given a local-graph sG of a target instance v, for each node $v' \in sG$, we first transform the initial node features $h^0_{v'}$ into embeddings at level $k = K$ as:

$$\begin{aligned} h^{k-1}_{N(v')} &= mean_pool\left(\{\mathbf{W}^{agg}_k h^{k-1}_u, \forall u \in N(v')\}\right), \\ h^k_{v'} &= sigmoid\left(\mathbf{W}^{proj}_k \cdot \text{concat}\left[h^{k-1}_{v'}, h^{k-1}_{N(v')}\right]\right), \end{aligned} \tag{2}$$

where $N(v')$ denotes the neighbors of node v', W^{agg} and W^{proj} denote the aggregation and projection matrices that are achieved by neural networks (neurons). At the end of the transformation (level K), we take the mean pooling of $h^{k=K}_{v'}$ across every node $v' \in sG$ to obtain the final local-graph feature vector s_t of the target instance v at time step t as:

$$s_t = \text{concat}\left[mean_pool\left(\{h^{k=K}_{v'}\}\right), tech(v)\right], \tag{3}$$

where $tech(v)$ denotes the technology features (from library files) of instance v in terms of driving strength, capacitance, and slew constraints of the current gate size. This vector s_t, which characterizes the local-graph and the underlying instance, is taken as the input of the RL-Sizer agent to determine the new gate size that helps improve the design performance. Note that the dimension of the GNN-encoded vector $h_{v'}$ is subject to the number of neurons in the last layer of the GNN module, which is 64 in our implementation.

D. Policy and Value Networks

We use DDPG [4], a variant of actor-critic algorithms, to build RL-Sizer. All actor-critic algorithms have two components that learn jointly: *actor* and *critic*. In deep RL (RL powered by neural networks), *actor* refers to the policy network which learns a parameterized policy $\pi(s)$ that maps a state vector s to an action a. Next, *critic* refers to the value network which learns a value function $Q(s, a)$ that evaluates the (discounted) reward of taking an action a on a state s.

Algorithm 2 RL-Sizer training methodology.

Input: Initial Policy Network parameters θ_π, Initial Q Network parameters θ_Q, Target networks update ratio ρ, Netlist $G = (V, E)$
Output: Policy Network parameters θ_π; Q Network parameters θ_Q

1: Initialize target networks (policy-, Q-) parameters $\{\phi\}$ as $\phi_\pi \leftarrow \theta_\pi$, $\phi_Q \leftarrow \theta_Q$, Replay Buffer $B \leftarrow \{\}$
2: **while** design TNS does not converge **do**
3: $\{V'\} \leftarrow$ instance_selection(G) ▷ Algorithm 1
4: $\{s\} \leftarrow$ local-graph_encoding(V') ▷ Equation 2, 3
5: $T \leftarrow |s|$ ▷ # of states (instances)
6: **for** $t = 0;\ t < T;\ t++$ **do** ▷ Assign actions for all cells
7: $a_t \leftarrow \pi(s_t|\theta_\pi)$
8: Perform actions $\{a\}$ and STA update to get rewards $\{r\}$
9: Store all (s_t, a_t, r_t, s_{t+1}) pairs in the replay buffer B
10: Sample a batch of T buffers $\{(s_t, a_t, r_t, s_{t+1})\}$ from B
11: **for** $t = 0;\ t < T;\ t++$ **do** ▷ Compute update targets y
12: $y_t \leftarrow r_t + \gamma * Q_{\phi_Q}(s_{t+1}, \pi(s_{t+1}|\phi_\pi))$
13: Update Q Network $\nabla_{\theta_Q} \sum_t (Q_{\theta_Q}(s_t, a_t) - y_t)^2$
14: Update Policy Network $\nabla_{\theta_\pi} \sum_t Q_{\theta_Q}(s_t, \pi(s_t|\theta_\pi))$
15: $\phi_\pi \leftarrow \rho\phi_\pi + (1-\rho)\theta_\pi$
16: $\phi_Q \leftarrow \rho\phi_Q + (1-\rho)\theta_Q$ ▷ Temporal difference update

Algorithm 2 presents the training process of RL-Sizer based on the DDPG [4] loss function update. In DDPG, the learning update of the Q-function Q is based on the Bellman equation, which suggests the Q-value $Q(s, a)$ at current state s to be computed in a dynamic programming manner as

$$Q(s_t, a_t) = \mathbb{E}\left[r_t + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})\right]. \tag{4}$$

In DDPG, the goal of the policy network π is to generate the action a_t subject to the state s_t that maximizes the Q-value $Q(s_t, a_t)$. The idea is that the higher the Q-value $Q(s, a)$ is, the better the action a is. The objective of the policy network π can thus be formulated as

$$\max_{\theta_\pi} E\left[Q(s_t, \pi(s_t|\theta_\pi))\right], \tag{5}$$

where $\pi(s_t|\theta_\pi))$ is the action output by the policy network π based on the encoded state vector s_t.

As shown in the algorithm, both the value and policy networks are trained by a technique named *temporal difference*

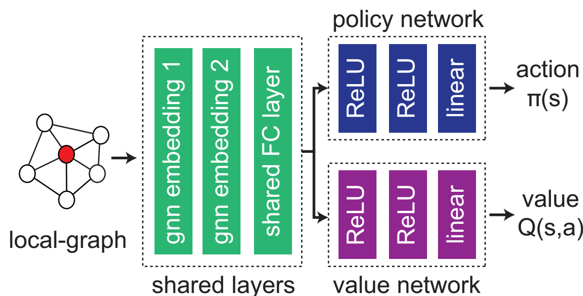

Fig. 3: Our RL agent architecture that consists of value and policy networks. Table II provides the dimension information.

TABLE II: Dimension of RL-Sizer layers.

component	input	hidden	output
shared layers	local-graph G=(V,E)	(64, 64) (GNN)	64 (FC)
policy network	64 (shared)	(64, 32) (ReLU)	1 (action)
value network	65 (shared + action)	(64, 32) (ReLU)	1 (value)

update, where for each network, we maintain a "target network" (with parameter ϕ) whose update is a trajectory slower than that of the main network (with parameter θ). For example, if the main network is updated in τ_i, then the the target network is updated in τ_{i+1}. By using a replay buffer B that contains old experiences from previous trajectories, the temporal difference update is expected to stabilize the training process. Finally, when the training completes, we obtain an *actor*, the policy network π, that performs the gate sizing moves to improve design performance. Figure 3 further shows the architecture of our RL agent that utilizes the value and policy networks.

E. Implementation Details: Challenges of ML in EDA

Our framework, RL-Sizer, is implemented in the source code of *Synopsys IC-Compiler II* (ICC2). Due to the fact that EDA tools are generally implemented in C++, while machine learning frameworks are mainly supported in Python, one of our main challenges is to communicate between these two language interfaces, since the communication introduces costly runtime overhead. Ideally, at each time step t of a sizing iteration, we can perform an action a_t on an instance and calculate the reward r_t immediately, so that instances in a common iteration will not interfere with each other. However, in our implementation, this ideal approach is not feasible considering the sizes of VLSI netlists. Even with the proposed technique of local-graph approximation, the runtime will still explode due to the communication overhead between C++ and Python. Therefore, to make the proposed framework practical, we only perform a full-chip STA update (i.e., switching from Python to C++) when all selected instances in a common iteration are assigned actions by the policy network (Lines 6–7 in Algorithm 2). After calculating all the rewards $\{r\}$ in the commercial tool, we again switch from C++ to Python and leverage gradient descent to update the network parameters.

IV. Experimental Results

In the experiments, we validate the proposed framework on 6 commercial designs (renamed due to confidentiality) in advanced technology nodes as shown in Table III, and demonstrate how RL-Sizer improves the native sizing algorithms in Synopsys ICC2, an industry-leading commercial tool.

TABLE III: Our commercial benchmarks and their attributes.

Design	Tech. Node	# Nets	# Macros	# Instances
block1	5nm	93,370	0	95,636
block2	5nm	145,893	0	151,258
block3	12nm	145,545	0	142,528
block4	12nm	430,141	35	462,755
block5 (SoC)	16nm	36,783	9	6,850
block6	16nm	72,748	0	71,604

A. Optimization Results

Table IV demonstrates the optimization results on our benchmarks as shown in Table III, where we observe that RL-Sizer can outperform or match the optimization results achieved by the reference commercial tool. Note that this is a head-to-head comparison, where the objectives of RL-Sizer and the commercial tool are exactly the same, which is optimizing design TNS through combinational sizing at the post-route stage. In RL-Sizer, we run Algorithm 2 for each design from scratch (i.e., the policy and value networks are trained from the beginning). We terminate the algorithm when design TNS no longer improves across 10 consecutive iterations. The results suggest that RL-Sizer is able to generalize across various designs and technology nodes. It also generalizes for both macro-heavy and macro-less designs.

Figure 4 further shows the design TNS after each sizing iteration (*RL trajectory*) of RL-Sizer on "block2" ($5nm$). Although the entire training process takes about 14 hours (250 iterations) to reduce TNS from -101.82ns to -0.81ns, it takes less then 3 hours (13 iterations) to quickly recover the initial TNS to -2.18ns. Note that the runtime in the table (and above) for both commercial tool and RL-Sizer is measured on the same machine without GPU support, and we do not limit the runtime of the commercial tool in order to perform thorough optimization (i.e., the tool stops the sizing optimization when the timing can no longer be improved). As for RL-Sizer, the stopping mechanism is aforementioned, and we expect the runtime to be significantly improved when GPUs are utilized.

B. Discussion of Optimization Results

The fact that RL-Sizer is able to perform commercial-grade timing optimization results on 6 different designs demonstrates its generality. However, we observe that RL-Sizer does not always outperform the commercial tool even though it adopts a more global approach to perform the optimization. This inferiority in fact happens on the designs that both the commercial tool and RL-Sizer are able to optimize the design TNS to "near zero". However, since RL-Sizer lacks rigid heuristics to "close design timing" as the commercial tool, the optimization results stagnate when no sizing action leads to positive reward. Ways to locally improve optimization results so as to completely close design timing are the areas for our future investigation.

Our further analysis reveal the following regarding the success of our RL-Sizer compared with the commercial tool:

TABLE IV: TNS optimization results comparison between RL-Sizer and Synopsys ICC2. The unit for timing is ns, and the unit for power is mW. WNS denotes worst negative slack; TNS denotes total negative slack, and #vio. EPs denotes the number of violating endpoints. The runtime for the commercial tool and RL-Sizer is measured on the same machine without GPU support.

Design (tech)	Initial (post-route stage)				Synopsys ICC2					RL-Sizer (ours)				
	WNS	TNS	#vio. EPs	total power	WNS	TNS (goal)	#vio. EPs	total power	run-time	WNS	TNS (goal)	#vio. EPs	total power	run-time
block1 (5nm)	-0.08	-61.99	2191	68.4	-0.08	-46.68	1728	68.7	30m	-0.07	-44.7	1631	68.8	6hr
block2 (5nm)	-0.05	-101.82	1305	204.4	-0.05	-1.19	182	205.3	1hr	-0.04	-0.81	116	205.7	14hr
block3 (12nm)	-0.31	-357.72	7934	44.2	-0.02	-0.07	4	44.9	1hr	-0.07	-0.37	39	45.1	15hr
block4 (12nm)	-0.22	-523.75	18845	123.6	-0.21	-8.71	348	123.9	1hr	-0.20	-8.11	201	124.1	22hr
block5 (16nm)	-0.80	-104.74	430	718.2	-0.79	-90.13	383	743.0	10m	-0.78	-80.54	379	718.4	5hr
block6 (16nm)	-0.15	-46.84	1377	25.2	-0.02	-0.03	5	25.5	24m	-0.04	-0.68	74	26.0	6hr

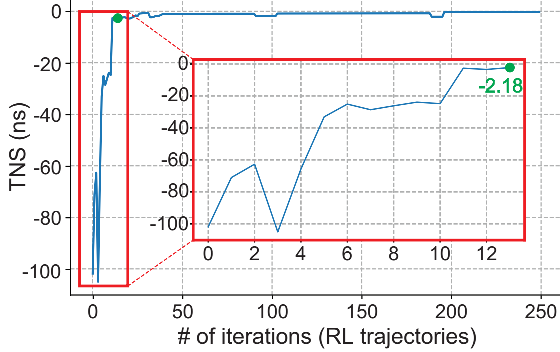

Fig. 4: RL-Sizer sizing iterations on block2 (5nm). It takes 250 iterations (about 14 hours) to improve TNS from -101.82 to -0.81 (ns). However, the TNS quickly converges in the first 13 iterations (less than 3 hours).

1) We accept "setback" moves: the goal of RL-Sizer is to maximize the total reward (i.e., sum of individual rewards) of a given iteration. Instead of striving to completely fix the entire slack violation for each selected instance, at some states, RL-Sizer learns to "setback" to create more sizing room for future states in order to achieve a global optima.
2) Our well-defined features: the initial node features in Table I accurately characterize the sizing behaviour of each instance. Despite these features are not sufficient to determine the final gate size of an instance, with the aid of the graph representation learning, they provide vital information for policy and value networks to effectively determine the sizes that optimize design performance.

Finally, we want to emphasize that although the main focus of this work is timing optimization, our framework can be extended to jointly optimize other PPA metrics such as power and area by incorporating them in the reward calculation (e.g., $r = \Delta_{timing} + \alpha * \Delta_{power} + \beta * \Delta_{area}$).

V. Conclusion and Future Works

Several prior works have made significant progress to improve VLSI gate sizing. In this paper, we take a new approach to solve the well-studied gate sizing problem using novel RL algorithms. We propose RL-Sizer, a GNN-powered policy gradient-based framework that performs automatic gate sizing for timing optimization without any human intervention. We believe the achieved optimization results shall demonstrate the great potentials of leveraging RL algorithms to solve classic EDA problems. In the future, we aim to validate the proposed framework on more designs and in various PD stages so as to enable transfer learning to accelerate the learning process. We will also investigate how to improve our local search heuristics so that RL-Sizer can completely close timing on able-to-close designs as the commercial EDA tools.

References

[1] C.-P. Chen, C. C. Chu, and D. Wong. Fast and exact simultaneous gate and wire sizing by lagrangian relaxation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(7):1014–1025, 1999.

[2] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In Advances in neural information processing systems, pages 1024–1034, 2017.

[3] H. Liao, W. Zhang, X. Dong, B. Poczos, K. Shimada, and L. Burak Kara. A deep reinforcement learning approach for global routing. Journal of Mechanical Design, 142(6), 2020.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[5] Y.-C. Lu, S. Nath, S. S. K. Pentapati, and S. K. Lim. A fast learning-driven signoff power optimization framework. In 2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).

[6] Y.-C. Lu, S. S. K. Pentapati, L. Zhu, K. Samadi, and S. K. Lim. 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. IEEE, 2020.

[7] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, et al. Chip placement with deep reinforcement learning. arXiv preprint arXiv:2004.10746, 2020.

[8] W. Ning. Strongly np-hard discrete gate-sizing problems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(8):1045–1051, 1994.

[9] G. Pasandi, S. Nazarian, and M. Pedram. Approximate logic synthesis: A reinforcement learning-based technology mapping approach. In 20th International Symposium on Quality Electronic Design (ISQED), pages 26–32. IEEE, 2019.

[10] H. Ren, G. F. Kokai, W. J. Turner, and T.-S. Ku. Paragraph: Layout parasitics and device parameter prediction using graph neural networks. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2020.

[11] S. Roy, D. Liu, J. Um, and D. Z. Pan. Osfa: A new paradigm of gate-sizing for power/performance optimizations under multiple operating conditions. In Design Automation Conference, page 129. ACM, 2015.

[12] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[13] H. Wang, K. Wang, J. Yang, N. Sun, H. Lee, and S. Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In ACM/IEEE 57th Design Automation Conference (DAC), pages 1–6. IEEE, 2020.

[14] Y. Zhang, H. Ren, and B. Khailany. Grannite: Graph neural network inference for transferable power estimation. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2020.