PIRLLS: Pretraining with Imitation and RL Finetuning for Logic Synthesis

Guande Dong*, Jianwang Zhai*,†, Hongtao Cheng, Xiao Yang, Chuan Shi, Kang Zhao†
Beijing University of Posts and Telecommunications, Beijing, China
{dongguande,zhaijw,zhaokang}@bupt.edu.cn

Abstract

As a key step in digital integrated circuit (IC) design, logic synthesis involves various logic optimization algorithms, where the quality of results (QoR) depends heavily on the optimization sequence used. Exploring the optimization space is challenging as the number of potential optimal permutations grows exponentially. Traditional methods rely on manual adjustments by experts, but are difficult to deal with complex and different circuits, leading to significant optimality gaps. Many automatic methods have been introduced, but still face problems of low generalization and low efficiency.

In this work, we propose PIRLLS, a two-stage learning framework for imitation learning on expert trajectories followed by reinforcement learning (RL) finetuning, to efficiently explore the optimal synthesis flows. Firstly, PIRLLS uses imitation learning to pretrain fast and high-performance intelligent policy, to fully leverage the offline knowledge of a large corpus of high-quality expert trajectories. Then, the pretrained policy is finetuned for target circuits using the proximal policy optimization (PPO) algorithm and policy distillation to obtain better results. Compared with the state-of-the-art (SOTA) method, our framework can effectively improve the quality of logic optimization and significantly speed up the exploration time.

ACM Reference Format:
Guande Dong*, Jianwang Zhai*,†, Hongtao Cheng, Xiao Yang, Chuan Shi, Kang Zhao†. 2025. PIRLLS: Pretraining with Imitation and RL Finetuning for Logic Synthesis . In *Proceedings of 30th Asia and South Pacific Design Automation Conference (ASPDAC 2025).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697786

1 Introduction

Logic synthesis converts a high-level circuit description at the register transfer level (RTL) into an optimized gate-level netlist, involving multiple stages such as translation, technology-independent optimization, and technology mapping. The goal of technology-independent optimization is to apply a series of optimization algorithms that reduce the depth and number of nodes in the circuit's Boolean network, thereby enhancing the circuit's performance and quality. ABC [1] is a popular open-source logic synthesis tool that utilizes an And-Inverter Graph (AIG) as its subject graph and provides heuristic synthesis flows like *resyn2* and *compress2* to optimize the structure of AIGs. Although this synthesis workflow is widely adopted, it can sometimes result in suboptimal results due to the different optimization strategies required by various circuits.

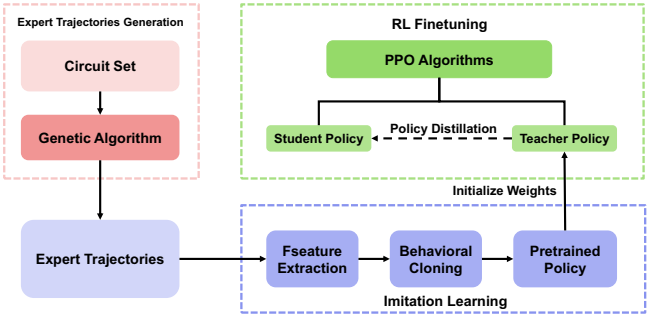


Figure 1: Illustration of PIRLLS Framework

As the demand for higher QoR continues to increase, machine learning (ML) techniques are used to classify or predict the final QoR of these synthesis flows, as documented in studies [2, 3], have been proposed. However, these prediction methods suffer from limited accuracy and are difficult to efficiently explore different circuits. Furthermore, people use reinforcement learning (RL) to conceptualize the generation of synthesis flows as Markov decision processes (MDPs) to autonomously generate synthesis sequences. The DRiLLS framework [4] introduces an advanced system enabling an A2C agent to flexibly select optimal transformations. Zhu *et al.* [5] employ Graph Neural Networks (GNNs) to capture the topology of AIGs and integrate it with historical decisions, thereby enriching the state information to enhance decision-making efficacy. Zhou et al. [6] apply random forests to conduct feature importance analysis, enabling feature pruning and investigating the generalization capabilities of RL within this context.

Recent studies have focused on developing synthesis flows tailored to specific circuits to enhance logic synthesis quality. Online learning frameworks such as CBTune [7] and AlphaSyn [8] showcase advanced optimization strategies. CBTune utilizes a contextual bandit approach with the Syn-LinUCB algorithm to efficiently explore the solution space and circumvent local optima by considering circuit characteristics and long-term impacts. Conversely, AlphaSyn employs a domain-specific Monte Carlo tree search (MCTS) with the SynUCT algorithm for precise selection and a parallel exploration process, showing significant improvements in area reduction and runtime over traditional methods.

Online learning methods can provide customized optimization, but face challenges in generalization across different scenarios. Most existing methods require fresh learning and exploration for each new circuit, lacking effective utilization of existing data and knowledge. Just like humans, we can learn offline knowledge from a large amount of existing data and use it to handle and improve new situations. Therefore, an ideal policy should consider how to learn general

*Co-first authors with equal contribution.
†Corresponding authors.

ASPDAC 2025, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01
https://doi.org/10.1145/3658617.3697786

offline knowledge from existing circuits and synthesis flows and finetune it for new target circuits.

This work proposes PIRLLS, a framework for pretraining via imitation and RL finetuning to facilitate logic optimization. PIRLLS uses a genetic algorithm to generate expert trajectories, pretrains policies via imitation learning, and fine-tunes them on target circuits with PPO and policy distillation, leveraging offline knowledge to facilitate exploration. The main contributions are as follows:

- To obtain high-quality offline knowledge, we design a genetic algorithm to efficiently generate a large number of expert trajectories for training circuits.
- To better characterize the circuit state, we use carefully designed scalar features and a pretrained BERT model to extract features from AIG and historical operations.
- We use behavior cloning to imitate expert trajectories and learn offline knowledge, and the pretrained policy can optimize unseen circuits rapidly and effectively.
- For more targeted optimization of target circuits, we use PPO and policy distillation to finetune the pretrained policy and maintain its advantage of efficient exploration.

2 Preliminaries

2.1 Logic Optimization and Technology Mapping

Logic optimization improves a circuit's efficiency and compactness by transforming and simplifying its logical representation, typically in the form of AIGs. Various algorithms like rewriting, balancing, refactoring, and resubstitution reduce the number of logic nodes and AIG depth, speeding up the circuit and lowering hardware requirements. Technology mapping converts the optimized network into a gate-level netlist using a specific technology library. Logic optimization directly influences the area and delay of the final netlist. This paper focuses on optimizing 6-input LUTs in FPGA using various ABC operators, exploring the optimization space through different combinations of operators.

2.2 Imitation Learning

Imitation Learning, or Behavior Cloning, is an ML technique where agents learn to perform tasks by mimicking expert behavior. This approach bypasses the need for a direct reward mechanism, instead using expert demonstrations to infer the mapping between observations and actions. By training on these demonstrations, the agent approximates the expert's decision-making process. AlphaGo [9], the first robot to defeat top human players, is pretrained using imitation learning on human game records to acquire human-level strategies. AlphaGo's success is largely attributed to its ability to learn from a vast database of historical Go games, further enhanced by reinforcement learning, which enabled AlphaGo to surpass human experts. Therefore, we exploit behavior cloning to learn offline knowledge from high-quality expert trajectories for pretraining to achieve efficient logic optimization.

2.3 RL and Policy Distillation

Reinforcement Learning is a key area of Machine Learning, involving states, actions, and rewards. Let $\tau = (s_t, a_t, r_t)$ represent a sequence of state, action, and reward tuples. The goal of RL is to find a policy $\pi_\theta(a \mid s)$ that maximizes the expected sum of discounted future rewards. At each time step t, the state s_t describes the environment's current conditions and serves as the input to the policy. The action a_t, chosen based on $\pi_\theta(a \mid s_t)$, affects the environment and leads to the next state. The reward r_t is received after taking action a_t and indicates how effective the action was. The policy $\pi_\theta(a \mid s)$ aims to select the optimal action a given the state s, with its parameters θ being adjusted through learning to improve the policy's performance. The optimal policy π^* is defined as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}[R_T], \quad \text{where } R_T = \sum_{t=1}^{T} \gamma^{t-1} r_t, \tag{1}$$

where γ is discount factor used to control future rewards, and T represents the length of the trajectory τ.

In practice, learning an optimal policy can be complex and computationally expensive. To address this, we employ Policy Distillation [10], a technique that transfers the knowledge of a complex policy model to another model. The primary goal is to train a student model to mimic a complex teacher model, minimizing the difference in their policy outputs. In our work, we leverage policy distillation to extract offline knowledge from the pretrained policy and explore target circuits in a more targeted manner.

2.4 BERT

BERT [11] is a pre-trained deep learning model based on the Transformer architecture. The model is trained on large amounts of text data and learns deep language features through two tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP). One of the core features of BERT is its bidirectional representation capability, which allows it to consider both the left and right context of each word simultaneously, producing richer and more accurate word embeddings. This makes BERT excel in various natural language processing tasks such as text classification, question answering, and language understanding. In this work, we use BERT to convert variable-length historical operator information into fixed-length vectors, helping the agent to better understand and utilize past operations, thus enhancing its decision-making capabilities.

3 Problem and Motivation

3.1 Problem

Various logic optimization algorithms, including balancing, rewriting, and refactoring, are used to optimize digital circuits. The arrangement and combination of different operators significantly affect the final optimization outcome, necessitating exploration within a vast design space. Defining $A = \{n_1, n_2, \ldots, n_m\}$ as the set of available optimizations in a logic synthesis tool and letting k be the length of optimized synthesis flows, there exist m^k possible flows, creating an exponentially large search space. This makes finding the optimal flow for complex circuit designs particularly challenging.

3.2 Motivation

Although optimal synthesis flows for different circuits may vary, useful knowledge can still be extracted from existing flows to aid new designs. For circuit designs that have certain similarities, their optimal flows are often also similar. We verified this experimentally, as shown in Table 1. We use a heuristic algorithm to find well-performing synthesis flows for similar circuits. The "Best" in Table 1 lists the top ten flows with the fewest LUTs per circuit. We then swapped these "Best" flows between similar circuits, shown as "Change" in Table 1, and compared them to the fixed "resyn2*2" flow. Results indicate that swapping flows between circuits leads to better optimization than using "resyn2*2". This suggests that leveraging historical synthesis data from numerous existing designs can enhance the optimization of new circuit designs.

Table 1: Synthesis of Structurally Similar Circuits

Circuit	Best	Change	resyn2*2
s838	59.9	63.5	65
s838.1	61.6	67	70
C7552	337.2	355.9	382
c7552	348.5	365.3	380
C3540	209	218.2	228
c3540	212.8	215	230
C2670	111.9	118	130
c2670	190	192	211.125
Average	191.36	199.4	211.1

Based on this analysis, we propose a two-phase method combining offline policy pretraining with online finetuning to optimize the synthesis flow of target circuits. In the offline phase, we use imitation learning to pretrain a policy that mimics expert trajectories, optimizing circuits that may have similar structures. While the offline policy generalizes well to unseen circuits, we further finetune it for the target design to achieve better optimization results.

4 PIRLLS Framework

4.1 Overview

As inferred in Section 3.2, the target circuit can benefit from similar existing circuits and synthesis flows. Based on this insight, we introduce PIRLLS, as illustrated in Figure 1. This method uses a genetic algorithm to explore circuits in the training set, generating optimal synthesis flows as expert trajectories. Importantly, it extracts effective circuit features and forms numerous (s_t, a_t) pairs, representing the "expert's" choice of optimization operator a_t at circuit state s_t. Imitation learning then pretrains the policy network on these pairs to learn offline knowledge with generalization. Due to circuit differences, the pretrained policies may not perform optimally, so we finetune them using reinforcement learning and policy distillation, where PPO's actor is initialized with pretrained weights.

4.2 Expert Trajectory Generation

To learn useful prior knowledge from existing designs, we need high-quality synthesis flows for each design. Therefore, to obtain the dataset for pretraining, instead of using existing synthesis flows, we design a fast genetic algorithm (GA) to generate a large number of high-quality expert trajectories. To achieve better logic optimization effects and greater optimization possibilities, we have selected 10 operators, including "rewrite", "rewrite -z", "balance", "refactor -z", "refactor", "resub", "resub -z", "dc2", "if -g", and "ifraig". The above operators are widely used in logic synthesis tasks to ensure generality.

The detailed algorithm is shown in Algorithm 1. The genetic algorithm abstracts candidate solutions as genes, defined by various operators, and measures fitness by the number of LUTs. The algorithm starts by generating an initial population and calculating each individual's fitness. Tournament selection follows, where a subset of individuals is selected randomly, and the one with the highest fitness becomes a parent for the next generation. This repeats until sufficient parents are chosen. Uniform crossover then allows parents to exchange genes at each position with a certain probability, creating new offspring and restoring the population to its original size. Mutation is applied to introduce diversity. Through iterations, less fit individuals are eliminated, enhancing the results significantly.

Algorithm 1 Expert Trajectory Generation

Input: G: Number of generations, P: Population size, C_p: Crossover probability, M_p: Mutation probability, C: Stopping condition for unchanged fitness
Output: Optimal synthesis flow

```
Initialize population with random gene values
for g = 1 to G do
    Evaluate the fitness of each individual
    if fitness unchanged for C consecutive generations then
        Break
    end if
    Select parents via tournament selection
    for i = 1 to P do
        Crossover selected parents with probability C_p
        Add offspring to new population
    end for
    for i = 1 to P do
        Mutate genes based on M_p
    end for
end for
```

Table 2: Scalar Features of AIG Circuits

ID	Feature	ID	Feature
1	Input	2	Output
3	Number of gates	4	Level
5	Width	6	Number of LUTs
7	Number of LUTs level	8	Avg of input node fan-out
9	Std of input node fan-out	10	Avg of and node fan-out
11	Std of and node fan-out	12	And_nodes percent
13	Not_nodes percent	14	And_node_reduction_percent

Multiprocessing improves the efficiency of fitness calculations. The optimal synthesis flow obtained for each design is determined as the "expert trajectory" and constitutes the pretraining dataset.

4.3 Pretraining with Imitation Learning

In Section 4.2, we introduced the method for generating expert trajectories. What is more important is how to learn offline knowledge from existing expert trajectories and obtain the high-quality pretrained policy network, which is achieved through imitation learning in this work. Unlike reinforcement learning, imitation learning trains the network to gradually maximize the probability of expert actions in the state-action probability distribution. This process involves two parts: feature extraction and behavior cloning.

4.3.1 Feature Extraction

As outlined in Section 2.3 on RL, characterizing the state of the AIG circuit for optimization is crucial. A feature extraction algorithm for AIGs was developed to better represent the circuit state for optimization, inspired by the feature importance analysis in RL4LS [6]. We identified 14 key scalar features, listed in Table 2, including input/output count, gate count, levels, width (maximum nodes per layer), mapped LUTs count and levels, average and standard deviation of fan-out for input nodes and all AND gates, ratio of AND to NOT gates, and AND gate reduction ratio. These features are normalized to improve training and inference efficacy.

In addition, historical operations optimized for AIG can also provide some state features. To better utilize this historical operation information, we use a pre-trained single-attention-layer BERT model

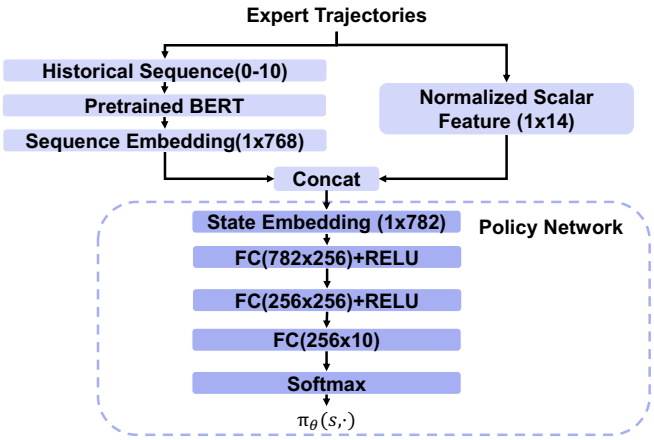


Figure 2: Feature extraction and poilcy network architecture

to capture these features. This choice is due to the variable length of historical operations, allowing BERT to convert this variable length into a fixed-length vector, thus capturing the contextual relationships within the optimization flow more effectively. Ablation studies have demonstrated the advantage of this approach. Then, these are concatenated with the normalized scalar features, as shown in the left part of Figure 2, ultimately forming our final features.

4.3.2 *Behavior Cloning*

We use Behavior Cloning (BC) to pretrain our policy based on expert trajectories. The policy network is pretrained to predict expert actions a_t from the state s_t obtained via the feature extraction network. The final state feature obtained from the scalar features and the BERT model is a 782-dimensional vector, which is then processed by three fully connected layers: the first maps it to 256 dimensions, the second is another 256-dimensional layer, and the third maps it to a 10-dimensional output. This output is then converted into an action probability distribution for actions using a softmax function.

Let $\pi_\theta^{\text{BC}}(a_t|s_t)$ denote a policy parametrized by θ that maps states s_t to a distribution over actions a_t. Let τ represent a trajectory consisting of state and action tuples, i.e., $\tau = (s_0, a_0, \ldots, s_T, a_T)$, and $T = \{\tau(i)\}_{i=1}^N\}$ denotes a dataset of expert trajectories.

The first loss function, L_{NLP}, is defined as the negative log-likelihood of the expert actions under the policy:

$$L_{NLP} = -\sum_{i=1}^{N} \sum_{(s_t,a_t)\in\tau^{(i)}} \log \pi_\theta^{BC}(a_t|s_t). \tag{2}$$

Minimizing L_{NLP} effectively maximizes the likelihood of the expert actions, aligning the policy outputs closely with the expert's decisions. Then, we introduce an entropy loss $L_{Entropy}$:

$$L_{Entropy} = -\sum_{i=1}^{N} \sum_{s_t\in\tau^{(i)}} \sum_{a\in A} \pi(a|s_t) \cdot \log \pi(a|s_t), \tag{3}$$

where $a \in A$ represents all possible actions. Minimizing $L_{Entropy}$ penalizes the certainty of the policy output distribution, making the probability distribution more uniform to ensure the generalization of the model. The total loss is defined as:

$$L_{BC} = L_{NLP} + \gamma L_{Entropy}, \tag{4}$$

where γ is a weighting factor. Through behavior cloning training, we obtain a pretrained policy that imitates the expert's behavior.

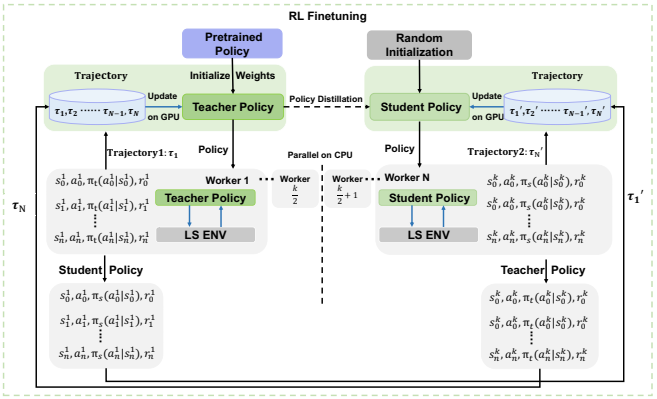


Figure 3: Distributed training for RL finetuning methodology

By incorporating mechanisms such as entropy regularization, the policy's generalization ability across different circuits is enhanced, ensuring robust decision-making in novel scenarios.

4.4 RL Finetuning

After imitation learning, we obtain a pretrained policy that can imitate the expert behavior to quickly and effectively optimize new circuits. However, since different circuits have variations, targeted adjustments need to be made according to the target circuit. Next, we use RL and policy distillation to finetune the pretrained policy.

4.4.1 *Reinforcement Learning*

To solve the maximization problem in Equation (1), we use an actor-critic RL approach, which consists of a learned policy (i.e., actor) and a state-value function $V(s)$ (i.e., critic). The critic $V(s_t)$ represents the expected value of returns R_t when starting from state s_t and acting under the policy π, where R_t defined as follows:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i. \tag{5}$$

In this work, we use Proximal Policy Optimization (PPO) [12], an on-policy actor-critic RL algorithm. PPO updates both the policy (actor) and the value function (critic) by utilizing advantage estimates. Specifically, the advantage estimate is calculated as:

$$\hat{A}_t = R_t - V(s_t), \tag{6}$$

and $p_t(\theta)$ is the ratio of the probability of action a_t under the current policy to that under the policy used to collect rollouts, defined as:

$$p_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|o_t)}. \tag{7}$$

The parameters are updated by maximizing:

$$J^{PPO}(\theta) = \mathbb{E}_t\left[\min(p_t(\theta)\hat{A}_t, \text{clip}(p_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right], \tag{8}$$

where ϵ is a small positive hyperparameter that restricts the policy update step, and the clip function ensures that the probability ratio $p_t(\theta)$ remains within the bounds $[1-\epsilon, 1+\epsilon]$, thus preventing destabilizing updates to the policy. More details about PPO can be found in [12]. The actor component of PPO refines policies by leveraging the stability provided by the clipped objective function. Additionally, the critic's value function reduces variance in policy updates, leading to more stable and efficient fine-tuning of the pretrained policy.

Our RL environment iteratively applies ABC optimization commands to the initial circuit netlist. The state and action spaces align with imitation learning. After each action, the environment provides

Table 3: PIRLLS vs. Baselines in 6-LUTs Optimization.

Benchmark	Initial	resyn2*2	DRiLLS [4]		RL4LS [6]		AlphaSyn [8]		CBTune [7]		Pretrain Only		PIRLLS	
	#LUTs	LUTs	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)
max	721	719	694	32.58	687.8	54.34	680	5.7	684.25	6.01	688	**0.183**	**675.6**	2.49
adder	249	249	244	24.05	244	10.05	244	6.14	244	5.97	211	**0.184**	**196.7**	1.62
cavlc	116	118	112.2	26.02	111.3	3.22	**106**	5.35	111	2.37	115	**0.155**	107.6	1.64
ctrl	29	29	28	24.25	28	2.85	28	5.68	28	0.59	29	**0.156**	**27**	1.66
int2float	47	46	42.6	21.7	42.3	2.81	**39**	5.54	40	2.76	42	**0.165**	**39**	1.54
router	73	76	70.1	22.01	69.5	3.07	65	5.34	68.11	2.32	72	**0.16**	**61.3**	1.64
priority	264	220	133.4	23.32	142.9	5.9	135	5.83	138.86	3.41	163	**0.193**	**126.3**	1.58
i2c	353	320	292.1	25.17	289.32	7.55	280	6.22	283.11	3.61	299	**0.165**	**271.3**	1.85
sin	1444	1466	1441.5	51.15	1438	20.1	1438	6.77	1441.67	9.71	1450	**0.235**	**1434.75**	3.57
square	3994	3915	3889.4	130	3889	72.88	3877	8.72	3882.11	25.99	3889	**0.39**	**3849.5**	7.92
sqrt	8084	5127	4708	147.64	4685.3	196.15	4415	9.83	4607	36.51	4038	**0.47**	**3837**	17.81
log2	7584	7703	7583.6	198.6	7580.1	125.28	7580	11.78	7580	41.27	7584	**0.61**	**7469.6**	12.8
multiplier	5678	5713	5678	180.84	5672	187.81	5672	10.34	5679.75	29.08	5675	**0.45**	**5663.6**	10.57
voter	2744	1828	1834.7	84.43	1678.1	330.48	**1537.4**	7.84	1682.25	11.46	1688	**0.28**	1579	3.7
div	23864	8197	7944.7	259.75	7807.1	482	6650.1	11.88	4180.91	25.58	4124	**0.48**	**3963.3**	17.52
mem_ctrl	11631	11459	10527.6	229.33	10309.7	1985.84	**9513.2**	10.99	10242.57	45.81	9838	**0.64**	9547.5	17.39
Average	4179.67	2949.06	2826.40	92.55	2792.20	218.14	2641.23	7.75	2555.84	15.78	2494.06	**0.31**	**2428.06**	6.58
Ratio	1.72	1.21	1.16	14.07	1.15	33.15	1.09	1.18	1.05	2.40	1.03	**0.05**	**1.00**	1.00

the next state and reward. Our goal is to find the optimal sequence to minimize the number of LUT-6. The reward function is:

$$r_t = \begin{cases} L_m - L(s_{t+1}), & \text{if } L(s_{t+1}) < L_m \\ -b, & \text{otherwise} \end{cases}, \quad L_{min} = \min(L_m, L(s_{t+1})), \tag{9}$$

where L_m records the best-found LUT-6 count up to time t and $L_m = L(s_0)$ initially. For each time step t, if $L(s_{t+1})$ is better than L_m, we give a positive reward corresponding to the improvement; otherwise, a small penalty $-b$ is applied. This reward function effectively encourages continuous improvement.

4.4.2 *Finetuning Methodology*

We initialize the actor's parameters with the pretrained policy weights π^{θ}_{BC}. Finetuning is challenging when the policy over-converges, skewing the actor's output towards a single action. To address this, we propose dual-policy exploration: the pretrained policy guides initial high-quality trajectories, while a randomly initialized actor enables broader exploration, balancing imitation and flexibility.

Due to the lengthy sample collection time, interaction with the environment becomes a training bottleneck. Therefore, we designed a distributed training mechanism using multiple processes to improve sample collection efficiency, as shown in Figure 3. Initially, we evenly distribute K environments between the pretrained "teacher policy" and the randomly initialized "student policy," gradually reducing the teacher's allocation over time. $\pi_s(a \mid s)$ denotes the student policy's action probability given state s, and $\pi_t(a \mid s)$ denotes the teacher's. During network updates, both policies utilize all collected trajectories (including those generated by the opposing policy), with each policy's network estimating the action probabilities for the states in the trajectories generated by the other. Additionally, both policies share a single critic to estimate the value function, thereby enhancing the consistency and efficiency of learning and policy performance.

To expedite the student policy's alignment with the teacher policy's performance, we utilize the principle of policy distillation and add the additional KL divergence loss L_{KL} to the standard PPO loss, i.e. L_{policy} and L_{value}. L_{KL} is defined as:

$$L_{\text{KL}} = \sum_{a \in A} \pi_s(a|s) \log \left(\frac{\pi_s(a|s)}{\pi_t(a|s)} \right). \tag{10}$$

This additional loss is intended to enable the student policy to rapidly achieve the performance levels of the teacher policy. Therefore, the student's loss is defined as:

$$L_{\text{student}} = L_{\text{policy}} + L_{\text{value}} + \alpha L_{\text{KL}}, \tag{11}$$

where α is a weighting factor. L_{policy} is mainly responsible for adjusting the policy to enhance the agent's performance in a specific environment by optimizing its behavior to maximize expected rewards. L_{value} aims to optimize the value function to more accurately predict future rewards, thereby helping to make better decisions under uncertainty.

To encourage exploration, we incorporate L_{Entropy} into L_{teacher}. L_{Entropy} is defined as in Equation (3), and the teacher's loss is as follows:

$$L_{\text{teacher}} = L_{\text{policy}} + L_{\text{value}} + \beta L_{\text{Entropy}}, \tag{12}$$

where β is a weighting factor. By employing this approach, the pretrained policy can be finetuned using RL and policy distillation. This allows for better utilization of offline knowledge to optimize specific circuits without compromising efficiency.

5 Evaluation

Comprehensive experiments are conducted to evaluate the proposed PRILLS framework. In Section 5.1, we first introduce the experimental settings. In Section 5.2, we give the optimization results and compare our method with previous work to prove its superiority. In Section 5.3, we perform an ablation study on PRILLS framework.

5.1 Experimental Settings

The PIRLLS framework is implemented in Python, using PyTorch and the RL framework OpenAI Gym [13]. In addition, the exploration environment of PIRLLS is implemented by developing C++ interfaces for ABC [1]. The experiments are conducted on Intel(R) Xeon(R) 8383 CPU @ 2.60GHz with an NVIDIA RTX 3090 GPU.

Table 4: LUT Results of Ablation Study

Benchmark	RL	RL+His	RL+BERT		Pretrain+RL+BERT	
	Last	Last	First	Last	First	Last
max	688	691	716	688.1	711	**685.6**
adder	203.1	201	216.6	200.7	211.1	**198.1**
cavlc	111.1	113.9	118.1	111.1	113.9	**110.2**
ctrl	28	28.1	28.8	28	28.2	**27.9**
int2float	43.9	43.4	43.9	**41.4**	43.4	41.7
router	66.5	65.4	70.5	64.4	66.8	**64.2**
priority	142.9	140.7	170.1	135.2	152.5	**127.9**
i2c	289	285.45	299.2	**272.9**	297	274.7
sin	1441.3	1439.7	1452	1440	1445.7	**1438.2**
square	3889	3886.7	3894.6	3862	3889	**3851**
sqrt	4692	4686	4249.7	3870.7	3936	**3847.4**
log2	7489.1	7584	7599.3	7479.1	7548.4	**7471**
multiplier	5687.6	5692	5687.6	5671.4	5692	**5661.6**
voter	1659	1672	1787	1632.4	1716.3	**1611.2**
div	4543	4436	4828	4386.3	4109.7	**4035.9**
mem_ctrl	10291.7	10267.1	10470.5	10274.6	10299.6	**9618.4**
Average	2579	2577	2601.9	2509.8	2516.2	**2441.5**

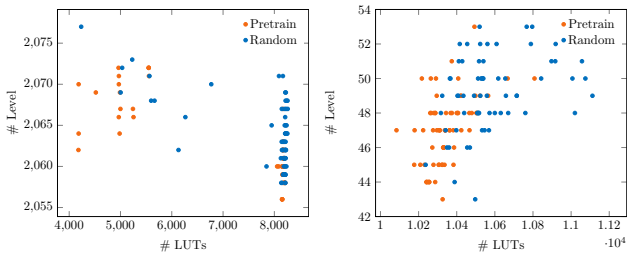
286 circuits from various well-known benchmark suites are used for pre-training, including ISCAS'85 [14], ISCAS'89 [15], ITC'99 [16], LGSynth'89 [17], LGSynth'91 [18], IWLS'93 [19], IWLS 2005 [20], and LEKO/LEKU benchmarks [21]. These benchmarks provide diverse circuits that guarantee the performance and generalizability of the pretrained policy network. We ran GA in Section 4.2 on these circuits to obtain expert trajectories. The settings of the GA are: population size of 100, gene length of 25, tournament selection (K=5), uniform crossover (35%), random mutation (10%), and 20 iterations. Although pretraining based on these expert trajectories using imitation learning takes time, this is a one-time cost, and LS users are more concerned about the time from design submission to synthesized output, i.e., the exploration time for target designs.

The number of LUTs (#LUTs) after technology mapping and the runtime (RT) are evaluated, comparing PIRLLS with baselines such as resyn2, DRiLLS [4], RL4LS [6], Alphasyn [8], and CBTune [7]. For the pretrained policy, actions are selected based on the highest probability for the test circuit state. PIRLLS is evaluated over ten experiments with different random seeds, averaging the results. To match the baselines, EPFL benchmarks [22] are used, sequence length is set to $L = 25$, and 'if -a -K 6' is applied for 6-LUT optimization.

5.2 Main Results

First, we test the pretrained policy obtained through imitation learning on 16 EPFL benchmarks. The policy directly selects actions to optimize the target circuit based on current state, with a sequence length of $L = 25$. As shown in Table 3, the pretrained policy ("Pretrain Only") significantly reduces optimization time and achieves high-quality results. Compared to DRiLLS [4], RL4LS [6], Alphasyn [8], and CBTune [7], our policy achieves 294×, 693×, 25×, and 51× speedup, respectively. The pretrained policy outperforms other baselines in optimization quality, demonstrating its effectiveness in accelerating optimization while maintaining high quality.

Although pretrained policiy obtained through imitation learning achieve competitive results, they may not perform optimally on new designs due to differences between circuit designs. Therefore, the proposed RL method is used to finetune the pretrained policy, forming the complete PRILLS framework. Experiments show that PRILLS can explore higher-quality optimization flows at meager runtime costs compared to other methods. As shown in Table 3, PRILLS outperforms DRiLLS [4], RL4LS [6], Alphasyn [8], and CBTune [7]



(a) div (b) mem_ctrl

Figure 4: Comparison between exploration based on pretraining and random exploration from scratch.

by 16%, 15%, 9%, and 5% in LUT count after technology mapping, respectively. More importantly, our framework leads in optimization time, achieving speedups of 14.07×, 33.15×, 1.18×, and 2.40×. Compared to SOTA CB-Tune, our framework achieves a 5% performance improvement in FPGA technology mapping with a 2.40× speedup.

The results show that pretraining the policy network with imitation learning and fine-tuning with multi-processing RL significantly improves optimization efficiency and quality.

5.3 Ablation Study

Ablation studies evaluate the effectiveness of our features and pretraining methods (Table 4). Each experiment ran 10 episodes with trajectories from 10 parallel sub-environments to update the policy and value networks. The "Last" column shows the average best #LUTs from the final episode, and the "First" column shows the initial episode averages. We compared "RL" (scalar features), "RL+His" (history operations), and "RL+BERT" (pretrained BERT). "Pretrain+RL+BERT" integrates the pretrained policy. The "Last" results confirm the superior performance of our full features.

To observe the impact of pretrained policies on RL, we selected two circuits: `div` and `mem_ctrl`. We compared the first episode optimization results of RL with and without the pretrained policy over 70 runs (Figure 4). The pretrained policy outperformed random exploration, offering better trajectories early in training and a superior starting point. Table 4 further confirms the effectiveness of pretraining, with a 3.3% improvement in the "First" phase and 2.7% in the "Last" phase for "Pretrain+RL+BERT" compared to "RL+BERT", highlighting the significant advantage of pretraining.

6 Conclusion

We propose PIRLLS, a two-stage learning framework for logic synthesis and optimization. First, we pretrain the exploration policy with imitation learning on expert trajectories, then finetune it on target circuits using reinforcement learning. Experimental results show PIRLLS surpasses SOTA methods across metrics, enhancing optimization quality and achieving significant speedup. In summary, PIRLLS leverages offline knowledge and customizes optimization for target circuits, utilizing expert trajectories from existing data and the self-improvement property of RL algorithms.

Acknowledgment

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404021), and the Beijing Natural Science Foundation (No. 4244107, QY24216).

References

[1] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference*, 2010, pp. 24–40.

[2] C. Yu, H. Xiao, and G. De Micheli, "Developing Synthesis Flows Without Human Knowledge," in *Proc. DAC*, 2018, pp. 1–6.

[3] A. Basak Chowdhury, B. Tan, R. Carey, T. Jain, R. Karri, and S. Garg, "Bulls-Eye: Active Few-Shot Learning Guided Logic Synthesis," *IEEE TCAD*, vol. 42, no. 8, pp. 2580–2590, 2023.

[4] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *Proc. ASPDAC*, 2020, pp. 581–586.

[5] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proc. MLCAD*, 2020, pp. 145–150.

[6] G. Zhou and J. H. Anderson, "Area-driven FPGA logic synthesis using reinforcement learning," in *Proc. ASPDAC*, 2023, pp. 159–165.

[7] F. Liu, Z. Pei, Z. Yu, H. Zheng, Z. He, T. Chen, and B. Yu, "CBTune: Contextual Bandit Tuning for Logic Synthesis," in *Proc. DATE*, 2024, pp. 1–6.

[8] Z. Pei, F. Liu, Z. He, G. Chen, H. Zheng, K. Zhu, and B. Yu, "AlphaSyn: Logic synthesis optimization with efficient monte carlo tree search," in *Proc. ICCAD*, 2023, pp. 1–9.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[10] A. A. Rusu, S. G. Colmenarejo, Çaglar Gülçehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, "Policy Distillation," *CoRR*, vol. abs/1511.06295, 2015.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL*, 2018, pp. 4171–4186.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *ArXiv*, vol. abs/1707.06347, 2017.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv: Learning*, Jun 2016.

[14] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *Proc. ISCAS*, 1985, pp. 677–692.

[15] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. ISCAS*, May 1989, pp. 1929–1934 vol.3.

[16] F. Corno, M. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *Design Test of Computers, IEEE*, vol. 17, no. 3, pp. 44–53, Jul 2000.

[17] S. Yang, "Logic Synthesis and Optimization Benchmarks," 1989 MCNC International Workshop on Logic Synthesis, Tech. Rep., Dec. 1988.

[18] ——, "Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0," Microelectronics Center of North Carolina (MCNC), Tech. Rep., Jan. 1991.

[19] K. McElvain, "IWLS'93 Benchmark Set: Version 4.0," a part of IWLS'93 benchmark set, Tech. Rep., May 1993.

[20] C. Albrecht, "IWLS 2005 Benchmarks," IWLS, Tech. Rep., Jun. 2005.

[21] J. Cong and K. Minkovich, "Optimality Study of Logic Synthesis for LUT-Based FPGAs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 230–239, Feb 2007.

[22] L. Amaru, P.-E. Gaillardon, and G. Micheli, "The EPFL Combinational Benchmark Suite," Jan 2015.