Logic Optimization Meets Deep Reinforcement Learning

Weihua Xiao
University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, China
Email: 019370910014@sjtu.edu.cn

***Abstract*—Logic optimization is a key step during the process of designing digit circuits, whose purpose is to reduce the area and delay of digital circuits. Several logic optimization algorithms have been proposed in the past decades, which have been developed as the basic logic optimization operators (LOO). Designers apply these LOOs iteratively and hybridly to reduce the area and delay of digital circuits continually. However, the order of applying these LOOs has a significant impact on the reward of logic optimization, which is designed by experience. In this project, we are aimed at determining the order of LOOs automatically with the aid of deep reinforcement learning (DRL). We model the logic optimization process, i.e., using LOOs iteratively, as a Markov decision process (MDP) with a special state representation and then we take advantage of Advantage Actor Critic (A2C) algorithm to solve this sequential decision problem. According to the experiment results, our DRL-based logic optimization achieves an improvement over the initial digital circuits by 10.02% on average and outperforms a traditional manually designed order of LOOs by2% on average.**

***Index Terms*—Deep Reinforcement Learning, Logic Optimization, Markov Decision Process, A2C**

I. Introduction

Logic optimization is an important intermediate step between the initial given logic networks and actual implementation for very large-scale integrated (VLSI) circuits design. In Fig. 1, it shows an example of the VLSI's design flow. Initially, users give some system specifications and then designers have to translate these into the initial design through some hardware languages such as verilog. As shown in this figure, the logic optimization do optimization to reduce the hardware consumption of the initial design. Before doing the final physical implementation, the design has to be translated such that the translated design consists of basic modules which can be realized by physical logic devices such as NAND, NOR, XOR, INV in application-specific integrated circuits (ASIC) case or look-up table (LUT) in field-programmable gate array (FPGA) case. In the past decades, several successful logic optimization algorithms have been proposed to reduce the hardware cost of the initial designs, which are develop as different LOOs. Moreover, these LOOs do optimization on designs following different basic theorems while they all represent the initial designs as logic graphs consisting of logic gates. To greatly take advantage of the optimization space of the initial design, traditional logic optimization process often take several steps to reduce the number of mapped devices and decrease the hardware's latency for increased speed. In each step, a LOO is selected to do optimization on the logic graph.

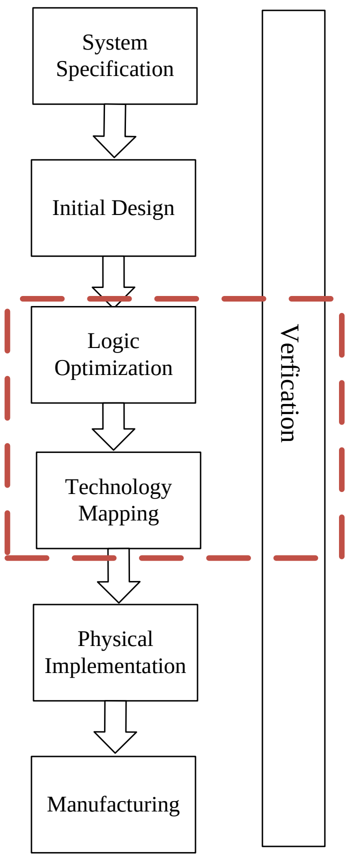

Fig. 1. The flow of designing integrated circuits.

A number of LOOs on the logic graph is able to reduce the hardware cost of the logic graphs without altering the logic function corresponding to the initial design. The reduction of the hardware cost behaves as reducing the number of nodes or depth of the logic graphs. After translating users' initial system specification to an optimized logic graph, the graph is mapped to the actual devices or gates in a specific hardware architecture, e.g., FPGA or ASIC. Finally, the logic gates are then programmed on FPGAs or physically placed and connected to generate a physical layout for manufacturing ASIC chips. ABC [1] is a well adopted tool to perform logic optimization and technology mapping in academia as shown in Fig. 1, in which there are several different kinds of LOOs, e.g. balance, rewrite and refactor.

Most of past works focused on designing new logic optimization algorithms to further explore the optimization opportunities on logic graphs. However, the order of applying these LOOs during logic optimization process has a significant impact on the final reduction of hardware cost, while the traditional works mostly determined the order of using these operators by heuristic or experience, such as resyn2 in ABC. In other words, there is still remaining optimization opportunity to obtain the optimal order of applying LOOs and we call this as the **LOO Order Optimization Problem**.

In this project, we focus on the LOO order optimization problem for FPGA. We model the logic optimization process as a MDP and apply a reinforcement learning (RL) algorithm to explore the search space for generating an effective order of LOOs. In the experimental part, we compare the achievement of our proposed order of LOOs with the initial designs. Moreover, to show the superiority of our work, we compare our proposed LOO order with ABC's heuristic *resyn2* to better results.

II. Preliminaries and Related Works

In this section, we discuss preliminaries.

A. Logic Graphs and Optimization

During the logic optimization process, the designs are oftern represented as a logic graph, which consists of basic logic gates. There are several different kinds of logic graphs, whose basic logic gates are different, such as And Inverter Graph (AIG) consists of AND and NOT gates, Majority Inverter Graph (MIG) consists of Majority and NOT gates. In our project, we select AIG as the graph representation, which is the most common representation in academia and can represent any combinational logic function. In Fig. 2, it shows an example of AIG. Before the final physical implementation step, the real area and delay of the logic graph can't be achieved and the two metrics are estimated by the number of nodes and levels in the graph. For exmaple, in Fig. 2, the area is estimated as 3 and the delay is 2.

After achieving AIGs of designs, the logic optimization tools such as ABC, can conduct logic optimization process on them. An example of logic optimization process is shown in Fig. 3, where the logic optimization tool is ABC. In each iteration, ABC selects one LOO o_i from $\{$*balance, refactor, refactor -z,resub,rewrite, rewrite -z*$\}$ to conduct optimization on the given AIG AIG_i and then it outputs the new AIG AIG_{i+1}.

B. Advantage Actor Critic

In this section, we introduce the necessary background of the reinforcement learning (RL). There are two kinds of algorithms in RL: value-based and policy-based. In the value-based DRL algorithms, it uses a neural network (NN) to train the value functions of states. After taking a new action a from the current state, s agents will achieve a new reward r and the next state s'. Then, the value of state s will be updated as $V'(s) = V(s) + \gamma * r$ and a new training sample

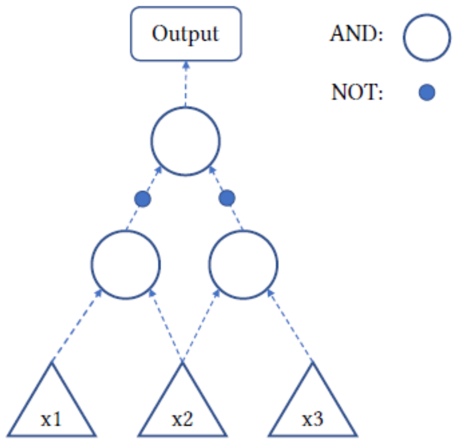

Fig. 2. An example of AIG.

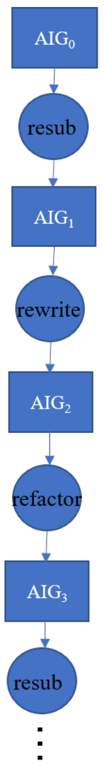

Fig. 3. An example of logic optimization process by ABC.

$(s, V'(s))$ can be given to the NN to update its parameters. By iteratively taking actions, the NN is trained continually and finally the agent achieves a great estimation of value functions. A representative value-based DRL algorithm is **Deep Q-learning Network (DQN)**. On the other hand, the policy-based DRL algorithms parametrize the policy as a continuous function π_θ and then formulate the policy search problem as a continuous optimization problem, which can be solved by a NN. A representative policy-based algorithm is **REINFORCE**.

The reinforcement learning algorithm: **Advantage Actor**

Critic (A2C) is a new type of DRL algorithm, which combines the advantages of value-based and policy-based algorithms. In A2C, it uses one network to train the state value functions and another to solve the policy search problem, which are so-called critic network and actor network, respectively. In our project, we take advantage of A2C to solve our Design Space Exploration (DSE) problem.

III. Methodology

In this section, we will introduce how to apply the A2C algorithm to solve the LOO order optimization problem for FPGA. Unlike most RL environments such as those in Gym [], where the environment is actually a game engine, the environment in our project is actually the logic optimization engine, i.e., ABC, over a given AIG. In other words, the states in Gym's environments are often easy to be represented and can interact with DRL algorithms friendly, such as images. However, the states in our case can't be directly obtained. In the following, we describe the constructing process of our designed RL environment, which has a friend interaction with A2C algorithm.

A. Action Space Representation

In each step of logic optimization, the agent, i.e., the A2C algorithm, selects one LOO and then the environment, i.e., ABC will act this LOO on the given AIG. Thus, a action is just one LOO and we select six LOOs from ABC and a null to construct the action spaces A as:**A={resub, refactor, refactor-z, rewrite, rewrite-z, balance, null}** The functions of the first five LOOs is to reduce the area of the given AIGs while the sixth LOO is able to reduce the delay, such that the delay and area of AIGs can be both improved during logic optimization process. The final null means that this step do nothing. This is also reasonable as considering runtime, the iteration times of logic optimization process is limited to an upper bound and sometimes we have achieved the largest rewards before meeting the upper bound of iteration times. Moreover, it will worsen the result if continuing to do logic optimization.

B. State Space Representation

At each time, ABC takes an action and generates a new AIG. Naturally, each state is corresponding to an AIG and the state space is represented as all possible AIGs processed during the logic optimization. However, the AIGs can't be processed by the critic network in A2C algorithm. The solution in our project is to extract features from an AIG which construct a feature vector, and we use this feature vector to represent the state regarding to each AIG. In this project, we extract these following features from an AIG:

$$\text{AIG state} = \begin{bmatrix} \text{\# primary I / O} \\ \text{\# nodes} \\ \text{\# edges} \\ \text{\# levels} \\ \text{\# latches} \\ \text{\% AND gates} \\ \text{\% NOT gates} \\ \text{\# MFFC nodes} \\ \text{levels-mean of delay distribution} \end{bmatrix}$$

The first five entries of this feature vectors shows several shallow structural features, such as the number of primary inputs and outputs of the AIG, the number of nodes, edges, levels and latches. The entries **%AND gates** and **%NOT gates** means the proportion of AND number and NOT number respectively, which in some sense represent the functional features. The last two entries are two heuristics for better fitting the states' value functions in critic network. From the lectures, the value functions of states represent the expectation of the sum of rewards and this means the expected reduction of area and delay of AIGs during the logic optimization process. Thus, it may be better to extract features of AIGs that have correlation with the reduction of area and delay. The maximum fanout free cone (MFFC) [2] is a special structure in AIGs, on which these area-oriented LOOs mainly do optimization such as refactor, refactor -z. In other words, this feature MFFC has a strong correlation with the area reduction during logic optimization. On the other hand, the last feature has a strong correlation with delay reduction. To keep the states within a specific range, as required by the agent's neural networks, we normalize all state values by their corresponding values for the initial input design.

C. Reward function Representation

In our logic optimization process, we attempt to reduce the area and delay of AIGs simultaneously. Thus, this is actually a multi-objective objective optimization problem. A direct approach is to make a weighted sum of the normalized area and delay. However, it is hard to set the weight as no idea to evaluate the importance between area and delay. In our project, the method is to transform it into a single objective optimization problem.

Our goal is to co-minimize the area and delay after FPGA technology mapping, i.e., the number of LUTs and levels of logic graphs, through logic optimization. We set an upper bound for one objective levels as the levels of the initial AIG after FPGA technology mapping. In other words, we only require that the levels are not larger than that of the initial AIG and put efforts on reducing the number of LUTs after mapping.

Table. III-C shows the reward formulation of this function. For each action and the delay of the current AIG is smaller

TABLE I
DEFINITION OF THE REWARD FUNCTION.

			Area		
			+	0	-
Delay Upper bound	$<$		-3	0	3
	$>$	+	-3	-2	-1
		0	-2	0	2
		-	1	2	3

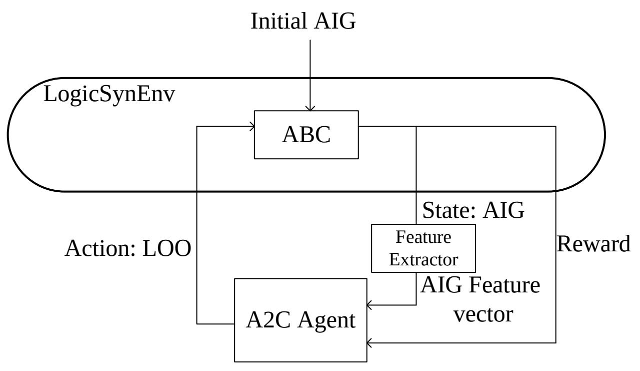

Fig. 4. The RL training framework with logic optimization environment.

than the delay upper bound: if the area of next AIG after FPGA mapping is larger than that of current AIG, then the reward is -3; if the area of next AIG do not change, then the reward is 0; if the area of next AIG is smaller, the reward is 3. On the other hand, for each action and the delay of the current AIG is larger than the delay upper bound, there are three different cases. We only make a detail explanation to one case ($+$, i.e., the delay of next AIG is increased) here: if the next AIG's area is larger, the reward is -3; if the next AIG's area has no change, the reward is -2; if the next AIG's area is smaller, the reward is -1. This reward strategy prevents the agent from receiving negative reward in all attempts in cases where the delay is larger than the upper bound. Moreover, when the area increases and the delay decreases (but not meeting the upper bound), a small positive reward is given as the agent is trying to learn from not meeting the constraint.

D. Training Framework

Util now, we have designed our own DRL environment–ABC, state representation, action space and reward function. Next, we only have to take advantage of the implemented DRL algorithm in the Stable-Baseline3 [3]. Thus, we do not illustrate the pseudo-code here, which can be achieved from our slides. Moreover, as mentioned before, we also give a limitation of the number of actions in a logic optimization process and thus our training is episodic with termination criterion as the number of steps. In Fig. 4, it illustrates the training framework with our designed ABC-based logic optimization environment.

IV. EXPERIMENTAL RESULTS

In this section, we will evaluate our DRL-based logic optimization on EPFL arithmetic benchmarks [4]. The technology mapping is conducted on AIG-based designs by ABC's command *if -K 6*. We will show the area and delay of optimized designs through our DRL-based logic optimization after FPGA technology mapping and compare these results with the initial designs and the optimized designs through a traditional order *resyn2* in ABC.

From Table. IV, we can clearly see that our propose DRL based logic optimization can achieve a 10.02% improvement over Area Delay Product (ADP) compared with the initial designs. Moreover, our automatic generated order of LOOs outperforms the heuristic order *resyn2* that can improve by 2% approximately on average. In other words, A2C's critic network indeed learns the value function, i.e., the expected rewards of logic optimization process of different AIGs and also A2C's actor network correctly generates a great policy.

REFERENCES

[1] A. Mishchenko, "Abc: A system for sequential synthesis and verification." [Online]. Available: https://people.eecs.berkeley.edu/ alanmi/abc/
[2] A. Mishchenko, R. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *ICCAD*, 2008, pp. 38–44.
[3] A. Hill *et al.*, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.
[4] L. Amarù, P. E. Gaillardon, and G. D. Micheli, "The epfl combinational benchmark suite," in *IWLS*, 2015.

TABLE II
AREA-DELAY COMPARISON OF LOGIC SYNTHESIS OPTIMIZATION RESULTS (ADP MEANS AREA DELAY PRODUCT).

Benchmark	Initial Deisign			Resyn2				Proposed			
	Area	Delay	ADP	Area	Delay	ADP	Improvement	Area	Delay	ADP	Improvement
Adder	254		12954	257	51	13107	-1.18%	253	51	12903	0.39%
	512	4	2048	512	4	2048	0.00%	512	4	2048	0.00%
		867	8072637	9831	858	8434998	-4.49%	6189	857	5303973	34.30%
		4194	187199190	44333	4191	185799603	0.75%	51374	4185	215000190	-14.85%
Log2	8008	77	616616	8060	71	572260	7.19%	8151	74	603174	2.18%
Max	842	56	47152	777	41	31857	32.44%	797	42	33474	29.01%
Multiplier	5913	53	313389	5916	53	313548	-0.05%	5913	53	313389	0.00%
Sine	1458	42	61236	1452	36	52272	14.64%	1456	36	52416	14.40%
Squre-root	5720	1033	5908760	4426	1005	4448130	24.72%	4425	1005	4447125	24.74%
Average							8.22%				10.02%