DRiLLS: Deep Reinforcement Learning for Logic Synthesis

Abdelrahman Hosny
Computer Science Dept.
Brown University
Providence, RI
abdelrahman_hosny@brown.edu

Soheil Hashemi
School of Engineering
Brown University
Providence, RI
soheil_hashemi@brown.edu

Mohamed Shalan
Computer Science Dept.
American University in Cairo
Cairo, Egypt
mshalan@aucegypt.edu

Sherief Reda
School of Engineering
Brown University
Providence, RI
sherief_reda@brown.edu

Abstract— Logic synthesis requires extensive tuning of the synthesis optimization flow where the quality of results (QoR) depends on the sequence of optimizations used. Efficient design space exploration is challenging due to the exponential number of possible optimization permutations. Therefore, automating the optimization process is necessary. In this work, we propose a novel reinforcement learning-based methodology that navigates the optimization space without human intervention. We demonstrate the training of an Advantage Actor Critic (A2C) agent that seeks to minimize area subject to a timing constraint. Using the proposed methodology, designs can be optimized autonomously with no-humans in-loop. Evaluation on the comprehensive EPFL benchmark suite shows that the agent outperforms existing exploration methodologies and improves QoRs by an average of 13%.

I. Introduction

Logic synthesis transforms a high-level description of a design into an optimized gate-level representation. Modern logic synthesis tools represent a given design as an And-Inverter Graph (AIG), which encodes representative characteristics for optimizing Boolean functions. Logic Synthesis mainly consists of three tightly-coupled steps, namely pre-mapping optimizations, technology mapping, and post-mapping optimizations. In the pre-mapping optimization phase, technology independent transformations are performed on the AIG to reduce the graph size resulting in a less total area, while adhering to a delay constraint. Next, in the technology mapping phase, the generic intermediate nodes are mapped to standard cells of a specific technology (e.g. ASIC standard cells). After that, post-mapping optimizations perform technology-dependent optimizations, such as up-sizing and downsizing.

Developing an efficient logic synthesis optimization flow proves to be an intricate task, requiring input from experienced designers. The complexity of designing such flows mainly arises from the exponentially large search space of the available transformations. In particular, with many transformation possibilities, different recurrence and permutations of such transformations can significantly affect the QoR [1], [2]. In addition, the increasing divergence and complexity in circuit designs have further complicated the design of optimization flows. It is crucial to note that there does not exist a predefined sequence of transformations that would generate best QoR for all possible circuits, and the optimization flows need elaborate tuning for each input.

At the same time, the advances in machine-learning (ML), and specifically reinforcement-learning (RL), have enabled autonomous agents to improve their capabilities in navigating a complex environment. Recently, successful implementations of such agents have shown to reach to human level or even outperform humans [3], [4]. For instance, AlphaGo was recently named the first computer to beat a professional human Go player [3].

In this light, we propose a novel methodology based on RL that aims at producing logic synthesis optimization flows. Our contributions in this work are as follows:

- We address the challenge of developing efficient design space exploration strategy. We map the problem of logic synthesis optimization to a game-like environment understandable by a reinforcement learning agent, and formulate a feature set extracted from the AIG characteristics. In addition, we derive a novel multi-objective reward function that aids the agent in minimizing the area subject to a delay constraint.
- We introduce DRiLLS (**D**eep **Rei**nforcemnet **L**earning-based **L**ogic **S**ynthesis), a novel framework based on reinforcement learning developed for generating logic synthesis optimization flows. Our methodology eliminates the need for a "human expert" tuning the synthesis parameters. It can be applied to any circuit design, without the need for a special setup.
- We demonstrate the capabilities of our proposed approach on the EPFL arithmetic benchmark suite [5]. We compare our work against best results from the benchmark suite when mapped to a standard cell library, in addition to classical optimization algorithms such as greedy heuristics. Expert-developed flows are also evaluated for baseline comparison. We show that DRiLLS outperforms previous techniques [5], [6].

The rest of the paper is organized as follows. First, in Section II, we define the problem and summarize relevant previous work. Next, in Section III, we present a background on RL that is utilized in our approach. Section IV provides the motivation for our work as well as a detailed discussion on the proposed methodology. After that, we summarize our experimental results in Section V. Finally, Section VI summarizes the main contributions of this paper and provides possible future directions.

II. Design Space Exploration

A. Problem Statement

In logic synthesis frameworks, there exist a rich set of primitive transformations, each optimizing the circuit using a different algorithm (e.g. balancing, restructuring). Permutations of these optimizations generate different QoR. Furthermore, different repetitions of the same transformations affect the QoR and therefore result in an **exponentially growing search**

978-1-7281-1220-7/20/$31.00 ©

space. Synthesis flows for large circuits often have tens or hundreds of optimization commands.

We define $\mathbb{A} = \{a_1, a_2, ...a_n\}$ as the set of available optimizations in a logic synthesis tool. Let k be the length of an optimization flow. Assuming that optimizations can be processed independently (e.g. no constraint for running a_1 before a_2), there exists n^k possible flows. Yu *et. al.* show that different flows indeed result in divergent area and delay results [1]. While human experts have traditionally guided the search, the increasing complexity of the designs and synthesis optimizations have highlighted the need for an autonomous exploration methodology.

B. Related Work

Methodologies for design space exploration (DSE) of computing systems and EDA technology have received significant interest in the research community. On architectural level, Ipek *et al.* propose predictive models based on neural networks, to explore the design space of memory, processor, and multi-chip processor domains and predict the performance [7]. Similarly, Ozisikyilmaz *et al.* explore design space pruning by performance prediction of different computing configurations [8]. In their work, they utilize three statistical models tuned on a small subset of the possible designs. A learning-based methodology, relying on random forests for design space exploration of high-level synthesis flows is also proposed [9].

More recently, Ziegler *et al.* proposed SynTunSys [2], a synthesis parameter tuning system which iteratively combines optimizations and focuses on the "survivor set" for further pursuit. Specifically, in each iteration, the candidates are assigned estimated costs and scenarios with the lowest cost values are evaluated. The cost estimator is then updated based on the learned costs [10]. Taking a different approach, Yu *et al.* mapped the problem of logic synthesis design flow composition to a classification problem [1]. They then utilize convolutional neural networks to classify sample flows, encoded as pictures, to "angel" or "devil" flows. Therefore, for their work they require a fixed length for the optimization, and a large sample size of pre-defined optimization flows for training and tests. Our work is different from the previous work in that we propose to use a reinforcement learning agent to explore the search space for the purpose of optimizing particular synthesis metrics (e.g., area and delay), and therefore, enabling variable length optimization flows, without requiring sample flows for training. Next, we discuss relative background on RL that is used in this work.

In recent years, reinforcement learning (RL) agents have demonstrated immense capabilities in navigating complicated environments [11], [3]. While earlier work using RL focused on domains with fully observable state space or where features could be handcrafted, Mnih *et al.* expanded these capabilities by introducing deep Q-networks (DQN) [11]. Capitalizing on recent advances in deep neural networks, their agent achieves state-of-the-art performance in comparison to previous models, performing comparable to humans. Further improving the capabilities of RL agents, in their work, Lillicrap *et al.* extended the action domain to continuous domain, targeting physical domains [12].

III. Background on Reinforcement Learning

In this section, we briefly discuss the background necessary for developing our methodology. In reinforcement learning, an agent is trained to choose actions, in an iterative manner, that maximize its expected future reward. Formally,

- At each iteration k, and based on the current state of the system s_k, the agent chooses an action a_k from a finite set of possible actions $\mathbb{A}$.
- With the application of the action at step k, the system moves to the next state s_{k+1} and a reward of $g(s_k, a_k)$ is then provided to the agent.
- The agent iteratively applies actions, changing the state of the system and getting rewards. It is then trained based on the collected experience to move toward maximizing its reward in future iterations.

A policy is defined as a mapping $\mathcal{M}$ that, for each given state, assigns a probability mass function $\mathcal{M}(\cdot|a)$ for an action [13]. There are two major categories for implementing the mapping $\mathcal{M}$: value-based and policy-based methodologies. In value-based methods (e.g. Q-learning) a value function is learned by the system that effectively maps $(state, action)$ pairs to a singular value [14], and picks the maximum over all possible actions. On the contrary, in policy-based methods (e.g. policy gradient), the optimization is performed directly on the policy ($\mathcal{M}$) [15]. Actor Critic algorithms [13], as a hybrid class, combine the benefits of both aforementioned classes.

In actor critic methods, a tunable critic network provides a measure of how good the taken action is (similar to a reward function), while the tunable actor network chooses the actions based on the current state. More formally defined, the actor policy function is of the form $\pi_\theta(s, a)$, and the critic function is of the form $\hat{q}_w(s, a)$; where s, and a represent the state and the action, while θ, and w represent the tunable parameters within each network. Therefore, there exist two sets of parameters, one for each network, that need to be optimized. The gradient optimization for the critic network is performed as,

$$\Delta w = \beta \delta \nabla_w \hat{q}_w(s_k, a_k) \tag{1}$$

where β sets different learning rate for policy and value. δ is the temporal difference error, which is defined as

$$\delta = R(s, a) + \gamma \hat{q}_w(s_{k+1}, a_{k+1}) - \hat{q}_w(s_k, a_k) \tag{2}$$

where γ is the discount factor. Similarly, the gradient optimization for the policy update (actor network) is then defined as

$$\Delta\theta = \alpha \nabla_\theta (\log \pi_\theta(s, a)) \hat{q}_w(s, a) \tag{3}$$

where α sets the learning rate. Note that actor network policy update is a function of the critic network as well, which allows it to take into consideration not only the current state of the environment, but also the history of learning from the critic network.

While very effective, actor critic models can suffer from high variability in action probabilities. Advantage functions are proposed as a solution to reduce this variability. The advantage function is defined as

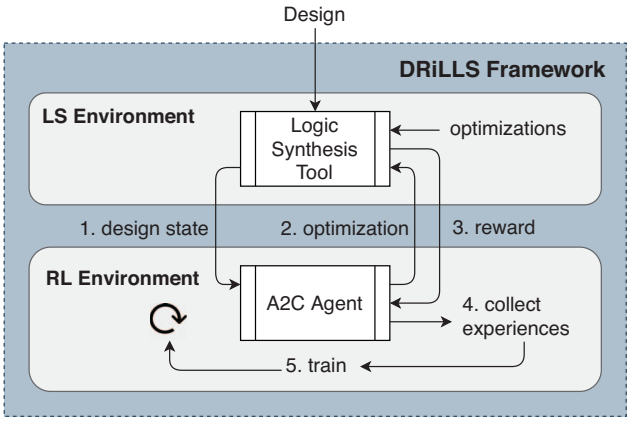


Fig. 1: The architecture of DRiLLS Framework. Numbers on the arrows represent the workflow of our methodology, and are illustrated separately in the subsections below.

$$A(s,a) = Q(s,a) - V(s) \tag{4}$$

where $Q(s,a)$ represents the Q value for action a in state s, and $V(s)$ represents the average value for the given state. In this work, we do not want to compute $Q(s,a)$. Instead, we formulate an estimate of the advantage function as

$$A(s) = r + \gamma V(s') - V(s) \tag{5}$$

where r is the current reward and γ is the discount factor. This achieves the same result without learning the Q function [16]. Next, we describe the proposed DSE methodology based on reinforcement learning.

IV. Methodology

DRiLLS, standing for **D**eep **Rei**nforcemnet **L**earning-based **L**ogic **S**ynthesis, effectively maps the design space exploration problem to a game environment. Unlike most reinforcement learning environments where gamification drives the behavior of the environment, the task here involves combinatorial optimization on a given circuit design. This makes it challenging to define the state of the game (i.e. environment) and the long-term incentives for an agent to explore the design space and not to fall into local minimums.

Figure 1 depicts the architecture of our proposed methodology. There are two major components in the framework: *Logic Synthesis* environment, which is a setup of the design space exploration problem as a reinforcement learning task, and *Reinforcement Learning* environment, which employs an *Advantage Actor Critic agent (A2C)* to navigate the environment searching for the best optimization at a given state. Next, we discuss both components and the interaction between them in details.

1. Design State Representation. In order to model combinatorial optimization for logic synthesis as a game, we define the *state* of the logic synthesis environment as a set of metrics retrieved from the synthesis tool on a given circuit design and used as a feature set for the A2C agent. As previously discussed, the *state* also represents the reaction of the environment to an *optimization* suggested by the second component of our framework, namely the *Agent*. Specifically, we extract the following state vector:

TABLE I: Formulation of the multi-objective reward function. ***Decr.*** stands for Decrease and ***Incr.*** stands for Increase.

			Optimizing (Area)		
			Decr.	*None*	*Incr.*
Constraint (Delay)	Met		+++	*0*	-
	Not Met	*Decr.*	+++	++	+
		None	++	0	- -
		Incr.	-	- -	- - -

$$\textbf{AIG state} = \begin{bmatrix} \#\ primary\ I/O \\ \#\ nodes \\ \#\ edges \\ \#\ levels \\ \#\ latches \\ \%\ ANDs \\ \%\ NOTs \end{bmatrix}.$$

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. Normalization is also a requirement for model generalization so that can be applied to unseen designs. On the one hand, optimizations change all elements in the *state* vector, except for the number of primary inputs and outputs. On the other hand, values in the *state* vector depict representative characteristics of the circuit. For example, a large $\#\ nodes$ value directs the agent towards reducing the number of nodes, which is achieved by restructuring the current AIG and maximally sharing the other nodes available in the current network (e.g. $resub$ and $refactor$ commands in ABC). Moreover, a large $\#\ levels$ value steers the agent towards choosing a $balance$ transformation. Hence, the *state* vector is representative of the circuit design at a given optimization step, and is aligned with the optimization space as we will discuss next.

2. Optimization Space. The agent explores the search space of seven primitive transformations, within ABC synthesis framework [17]. Specifically, $\mathbb{A} = \{$resub, resub -z, rewrite, rewrite -z, refactor, refactor -z, balance$\}$. The first six transformations target size reduction of the AIG, while the last one (balance) reduces the number of levels. These transformations manipulate the *state* vector representation discussed above, and are appropriate for the reward function illustrated next.

3. Reward Function. We define a multi-objective reward function that takes into account the change in both design area and delay. In particular, the agent is rewarded for reducing the design area, while keeping the delay under a pre-specified constraint value. Table I shows the reward formulation of this function. For each metric (design area or delay), a transformation would decrease, increase or make no change to the metric. Accordingly, we give the highest reward (represented as +++) for a transformation performed on a given AIG state that reduces the area and meets the delay constraint. We give the lowest negative reward when the transformation performed increases the design area and delay, while not meeting the constraint. Between the two extremes, the values and magnitudes of the reward have been chosen carefully to aid in the agent exploration. Essentially, we prioritize meeting the delay constraint. When not met, a positive reward is also given if the delay improved (i.e. decreased). This reward strategy prevents the agent from receiving negative reward in all attempts in cases where the delay constraint was too tight for the design to meet. Moreover, when the area increases and the delay decreases (but not meeting the constraint), a small positive reward is given as the agent is trying to learn from not meeting the constraint. This reward formulation has proved to be efficient as we will discuss in the next section.

4. Collecting Experiences. Algorithm 1 summarizes the operation of our proposed methodology. Here, lines 1 and 2 initiate the logic synthesis environment and the agent, respectively. Next, the agent is

Algorithm 1: DRiLLS Framework

```
   Input  : Design, Primitive Transformations
   Output: Optimization_Flow
1  env = Initialize(LS_Env);
2  agent = Initialize(A2C);
3  for episode = 1 to N do
4      episode_design_states = [];
5      optimization_sequence = [];
6      synth_rewards = [];
7      design_state = env.reset();
8      for iteration = 1 to k do
9          opt_probs = agent.ActorForward(design_state);
10         primitive_opt = RandomChoice(opt_prob);
11         [next_design_state, synth_reward] =
           env.perform(primitive_opt);
12         episode_design_states.append(design_state);
13         optimization_sequence.append(primitive_opt);
14         synth_rewards.append(synth_reward);
15         design_state = next_design_state;
16     end
17     episode_rewards = DiscountRewards(synth_rewards,
         gamma)
18     loss = agent.OptimizerForward(episode_design_states,
         optimization_sequence, episode_rewards);
19     agent.update(loss);
20     log(episode);
21 end
```

trained over the span of N episodes, where in each episode the logic synthesis environment is restarted; i.e. the original input design is reloaded (line 7). Next, in lines 8-16, the agent iteratively suggests a sequence of k primitive optimizations to produce the optimization flow. More specifically, first, in line 9, the agent computes the probability distribution of choosing one primitive optimization from the optimization space, $\mathbb{A}$. Then, in line 10, one of the primitive optimizations is selected according to the probability distribution calculated in line 9. Next, in line 11, the selected optimization is executed to determine its effect on the *design_state*. In addition, the reward is computed using the reward function in Table I. After that, we store the synthesis state, the optimization performed and the reward in the pre-initialized variables. Finally, we transition the state of the agent to the state after performing the optimization. The number of iterations is capped by k to provide the game with an elimination condition, and as the optimization improvements on a given circuit design fade out in later iterations. After all iterations are performed, we train the A2C agent from the collected experiences as we will discuss next.

5. A2C Agent Training. The training step starts with discounting the delay rewards over iterations in order to give earlier iterations a higher priority in choosing a good optimization (line 17). After that, in lines 18-19 the loss is computed and the actor and critic networks are trained to minimize the loss value as described next. As discussed in Section III, the agent has a hybrid policy-based and value-based networks, called actor and critic respectively. Both networks have an input layer of size equal to the *AIG state* vector length. In addition, a reward, r is passed to the critic network for training, and a discounted reward is passed to the actor network (Equation 5). The actor network outputs probability distribution over the available transformations. Therefore, the output layer in the actor network has a size equal to the size of $\mathbb{A}$. Since the agent is initialized with random parameters, transformations chosen in the start of the training process do not necessarily represent a good choice. Parameters of both networks are updated to reduce the loss using a gradient-based optimizer. This process is then repeated for a pre-defined number of times (called episodes), during which the agent is trained to predict improved optimization flows. In fact, the choice of a hybrid reinforcement learning architecture is suited for combinatorial optimization tasks as it gives the agent an opportunity to explore diverse optimization sequences, yet maintain a path towards optimal designs.

V. Experimental Results

We demonstrate the proposed methodology by utilizing the open-source synthesis framework ABC v1.01 [17]. We implement DRiLLS in Python v3.5.2 and utilize TensorFlow r1.12 [18] to train the A2C agent neural networks. All experiments are synthesized using ASAP7, a 7 nm standard cell library in typical processing corner. We evaluate our framework on EPFL arithmetic benchmarks [5], exhibiting wide ranges of circuit characteristics. The characteristics of the evaluated benchmarks (e.g. I/Os, number of nodes, edges and levels) can be found in [5]. Experimental parameters were setup as:

- **Episodes (N):** 50, **Iterations (k):** 50
- **Networks Size:** *Actor:* 2 fully connected layers, 20 hidden units each. *Critic:* one hidden layer with 10 units.
- **Weight initialization:** Xavier initialization [19]
- **Optimizer:** Adam [20], **Learning Rate:** (α): 0.01
- **Discount rate (γ):** 0.99

A small number of layers is used as we observe that deeper neural networks exhibit a random behavior and do not train well in this framework. This is attributed to the nature of the small number of features and transformations used. The experimental results are obtained using a machine with Intel Xeon 2x14cores@2.4 GHz, 128GB RAM, and 1x500GB SSD; running Ubuntu 16.04 LTS. Next, we present our results.

A. Design Space Exploration

Figure 2 shows traces of the agent searching for an optimized design that minimizes area, and meets the delay constraint. We plot one episode that finds the global minimum for a number of representative benchmarks. Generally, Figure 2 shows the attempts of the agent to balance between reducing the design area and meeting the delay constraint. For example, we observe the various trials of the agent to execute a transformation that reduces the delay to meet the constraint, but increases the design area such as iteration 30 in Log2 and iteration 26 in Max. Occasionally, exploration saturates as we can notice near-straight lines in some iterations. This shows the ability of the actor-critic networks to guide the exploration, while occasionally exploring other transformations that might open new search paths.

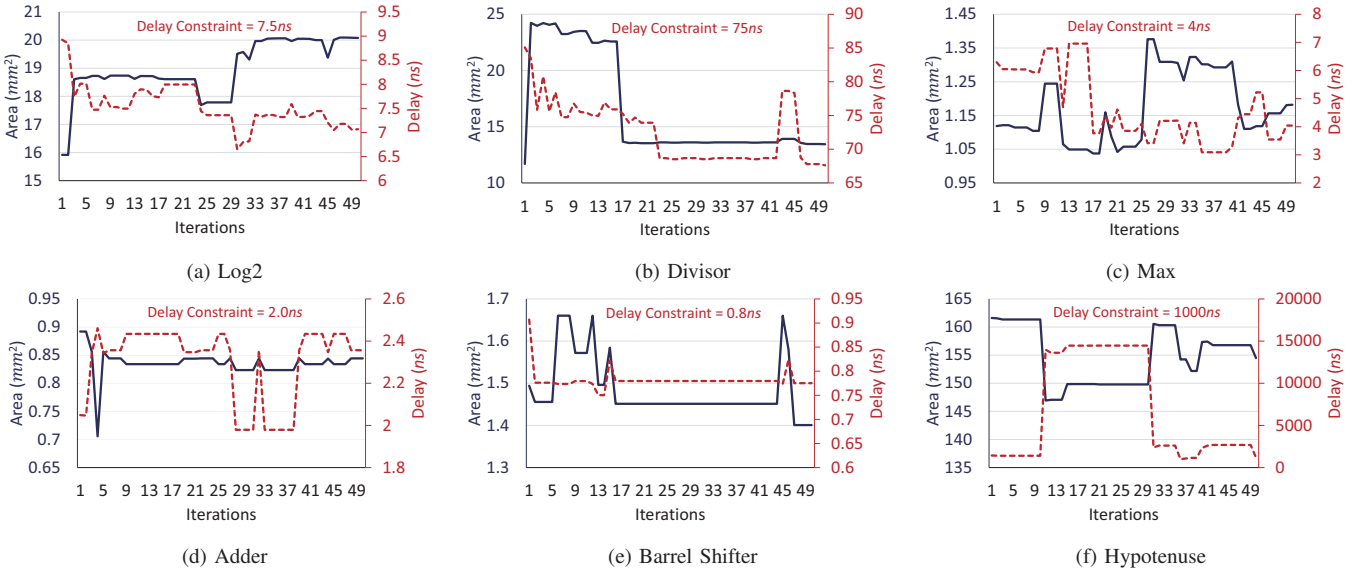
B. Comparison to Other Techniques

We compare the agent's performance against *EPFL best results*, *expert-crafted scripts*, and a *greedy heuristic algorithm*:

1) *EPFL best results*: best results are provided for size and depth. We compare against best results for size, since it is more relevant to the agent's nature of optimizing for area when mapping to a standard cell library.
2) *Expert-crafted scripts*: we maintain a record of expert-crafted synthesis optimizations derived from [6].
3) *Greedy heuristics algorithm*: we developed a baseline comparison that takes an initial input design and spawns parallel threads to perform each of the given AIG transformations on the design. Afterwards, each thread performs the mapping step using the delay constraint. The algorithm then evaluates the mapped designs from all threads, and keeps the one with the minimum area for the next iteration. After that, the process is repeated until two iterations yield the same area.

Table II gives the results of the mentioned comparisons. The area and delay for the initial design are obtained by loading the non-optimized designs in ABC and mapping them to ASAP7 without performing any transformation on the AIG. The delay is reported using the built-in timer in ABC (using *stime* command). We use the initial run to select a delay constraint value that challenges all the methods studied in this work. We make the following observations:

- The greedy algorithm has a single optimization target (area). Although the delay constraint was met in 4 designs, it is attributed to the best-effort mapping step that considers the delay constraint. The increase in the area occurs in the first



(a) Log2 (b) Divisor (c) Max

(d) Adder (e) Barrel Shifter (f) Hypotenuse

Fig. 2: Traces of DRiLLS agent navigating the design space to find a design with a minimum area while meeting the delay constraint.

TABLE II: Area-delay comparison of logic synthesis optimization results. A greedy algorithm optimizes for area. Expert-crafted scripts are derived from [6]. EPFL best results for size are available at [5].

Benchmark	*Delay*	Initial Design		Greedy			Expert-crafted [6]			EPFL Best Size [5]			DRiLLS		
	Constr.	*Area*	*Delay*	*Area*	*Delay*	*Impr.*	*Area*	*Delay*	*Impr.*	*Area*	*Delay*	*Impr.*	*Area*	*Delay*	*Impr.*
	(ns)	*(um^2)*	*(ns)*	*(um^2)*	*(ns)*	*(%)*	*(um^2)*	*(ns)*	*(%)*	*(um^2)*	*(ns)*	*(%)*	*(um^2)*	*(ns)*	*(%)*
Adder	2.00	867	2.02	1011	4.10	-16%	1772	1.82	-104%	1690	1.87	-94%	823	1.97	5%
B. Shifter	0.80	2499	1.03	2935	0.66	-17%	1534	0.77	38%	1040	0.77	58%	1400	0.77	43%
Divisor	75.00	12388	75.83	22439	79.14	-81%	21167	65.05	-70%	16031	74.91	-29%	13441	67.61	-8%
Hypotenuse	1000.00	176938	1774.32	236271	563.12	-33%	210828	525.34	-19%	169468	1503.88	4%	154227	995.95	12%
Log2	7.50	19633	7.63	30893	6.96	-57%	18451	7.45	6%	23999	10.12	-22%	17687	7.44	9%
Max	4.00	1427	4.48	3082	3.79	-115%	1440	3.93	-0.88%	1713	4.84	-20%	1037	3.76	27%
Multiplier	4.00	19617	3.83	25219	4.38	-28%	21094	3.70	-7%	19940	5.27	-1%	17797	3.96	9%
Sin	3.80	3893	3.65	5501	2.88	-41%	4421	2.19	-13%	4892	4.14	-25%	3050	3.76	21%
Square-root	170.00	11719	329.46	19233	93.71	-64%	16594	92.30	-41%	9934	169.46	15%	9002	167.47	23%
Square	2.20	11157	2.27	19776	3.96	-77%	16373	1.59	-46%	16838	4.06	-50%	12584	2.199	-12%
Avg. Area Imprv.		0.00%		-53.31%			-26.00%			-16.69%			13.19%		
Constraint Met		2/10		4/10			9/10			4/10			10/10		

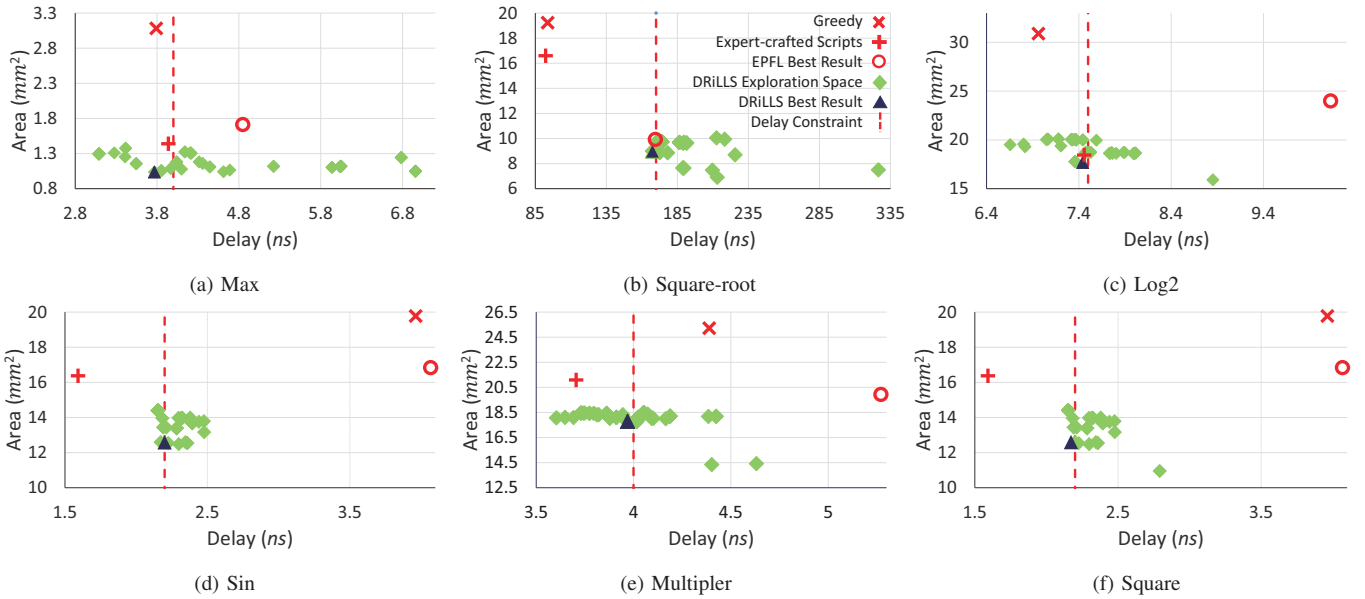
iteration that tries to meet the delay constraint while mapping. Since the algorithm meets the stop criteria in the first few iterations, it fails to reduce the area subject to a delay constraint. Results show the smallest average area improvement.

- Although expert-crafted synthesis scripts have not improved the designs' areas, they produced optimized designs that meet the delay constraint in 9 out of 10 designs. This comes at no surprise as the techniques used strive to meet the delay constraint; therefore, accepting near-optimal area results [6].
- EPFL best results have shown decent improvements in 3 designs, meeting the delay constraint in 4 of them. Although we benchmarked on the best results in terms of size, not depth, it is reasonable that their optimization techniques have not been designed for standard cell library mapping.
- DRiLLS agent meets the delay constraint in all designs while simultaneously improving the design area by an average of 13.19%. In the two designs that DRiLLS increased their area, it in fact met the delay constraint which the un-optimized design did not meet. This proves that the reward function defined before is an effective one for training the agent. Moreover, DRiLLS outperforms EPFL best result in all designs except *Barrel shifter*.

In the interest of space, Figure 3 elaborates on Table II by plotting the area-delay trade-offs offered by DRiLLS against the greedy algorithm, the expert-crafted synthesis scripts and the EPFL best results on six of the benchmarks. We define the **exploration run time** as the total run time of the agent, including interacting with the *Logic Synthesis Environment*, extracting AIG characteristics, and optimizing the parameters of the agent networks. The smallest design (Adder) is explored in *3.25mins*, while the largest (Hypotenuse) is explored in *25.46mins*. The average exploration time is *12.76mins* per episode. It is important to note that a trained model on one circuit design can be used (reloaded) into a new exploration on new circuits requiring no retraining.

VI. Conclusions

The goal of developing DRiLLS is to offer an autonomous framework that is able to explore the optimization space of a given



(a) Max (b) Square-root (c) Log2

(d) Sin (e) Multipler (f) Square

Fig. 3: Design area vs. delay trade-offs. The vertical dotted red line shows the delay constraint. For each benchmark, DRiLLS exploration space is indicated in green diamonds. A highlighted triangle represents the best optimized design that meets the delay constraint. Other methods are shown in red color with a cross mark, a plus mark and a circle for greedy, expert-crafted and EPFL result respectively.

circuit design, and produce a high Quality of Result (QoR) with no human in-loop. The intuition behind modeling this problem into a reinforcement learning context is to provide the machine with a methodology to try and error, similar to how human experts gain their experience optimizing designs.

In this work, we have presented a methodology based on reinforcement learning that enables autonomous and efficient exploration of the logic synthesis design space. Our proposed methodology maps the complex search space to a "game" where an advantage actor critic (A2C) agent learns to maximize its reward (reduce area subject to a delay constraint) by iteratively choosing primitive transformations with the highest expected reward. We have formulated an AIG state representation that has proved to effectively represent the feature set of a design state. In addition, we have introduced a novel multi-objective reward function that guides the exploration process of the agent. It allows the agent to find a minimum design area subject to delay constraint. Evaluating ten representative benchmarks, our proposed methodology manifests results that outperform existing methods.

DRiLLS proves that Reinforcement Learning can be used in combinatorial optimization of hardware circuit designs. It has a broad potential to be applied on related physical synthesis tasks, eliminating the need for human experts. The framework is open-source under a permissive license (BSD-3) and is available publicly on GitHub[1].

Acknowledgments

This work is supported by DARPA (HR0011-18-2-0032).

References

[1] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Design Automation Conference*, ser. DAC '18. ACM, 2018, pp. 50:1–50:6.

[2] M. M. Ziegler, H.-Y. Liu *et al.*, "A synthesis-parameter tuning system for autonomous design-space exploration," in *DATE*, 2016, pp. 1148–1151.

[3] D. Silver, J. Schrittwieser *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[4] M. Jaderberg, W. M. Czarnecki, Dunning *et al.*, "Human-level performance in first-person multiplayer games with population-based deep reinforcement learning," *arXiv preprint arXiv:1807.01281*, 2018.

[1] https://github.com/scale-lab/DRiLLS

[5] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *IWLS*, no. CONF, 2015.

[6] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *ICCAD*. IEEE, 2012, pp. 597–604.

[7] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," *SIGPLAN Not.*, vol. 41, no. 11, pp. 195–206, Oct. 2006.

[8] B. Ozisikyilmaz, G. Memik, and A. Choudhary, "Efficient system design space exploration using machine learning techniques," in *45th ACM/IEEE Design Automation Conference*, June 2008, pp. 966–969.

[9] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Design Automation Conference*, May 2013, pp. 1–7.

[10] M. M. Ziegler, H.-Y. Liu, and L. P. Carloni, "Scalable auto-tuning of synthesis parameters for optimizing high-performance processors," in *ACM International Symposium on Low Power Electronics and Design*, 2016, pp. 180–185.

[11] V. Mnih, K. Kavukcuoglu *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[12] 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.

[13] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[14] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[16] V. R. Konda and J. N. Tsitsiklis, "Onactor-critic algorithms," *SIAM journal on Control and Optimization*, vol. 42, no. 4, pp. 1143–1166, 2003.

[17] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*, pp. 1–17, 2007.

[18] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.