AlphaSyn: Logic Synthesis Optimization with Efficient Monte Carlo Tree Search

Zehua Pei[1], Fangzhou Liu[2], Zhuolun He[1], Guojin Chen[1], Haisheng Zheng[2], Keren Zhu[1], Bei Yu[1]
[1]Chinese University of Hong Kong [2]Shanghai Artificial Intelligence Laboratory

***Abstract*—Recent years have seen rising research in logic synthesis recipe generation to improve the Quality-of-Result (QoR). However, existing approaches typically have low efficiency and are stuck at local optima. In this work, we propose a logic synthesis optimization framework, AlphaSyn, that incorporates a domain-specific Monte Carlo tree search (MCTS) algorithm. AlphaSyn enables exploration across the entire search space while optimizing sampling points utilization. We further develop a synthesis-specific upper confidence bound for trees (SynUCT) algorithm for the selection phase and a well-designed learning strategy to enhance the stability of the MCTS algorithm. The AlphaSyn algorithm is fully parallelized for efficiency with asynchronous MCTS exploration and significance-base resource allocation. For standard-cell technology mapping on the ASAP 7nm library among other tasks, experimental results show that AlphaSyn outperforms SOTA FlowTune with an average 8.74% area reduction and 1.24× runtime speedup.**

I. Introduction

Logic synthesis bridges the register-transfer level and the optimized gate-level of circuit design through a sequence of synthesis transformations applied iteratively for logic optimization. The sequence of transformations in logic synthesis is typically obtained either from the heuristic scripts offered by synthesis tools or constructed based on prior knowledge. However, this current approach suffers from a significant optimality gap due to the fact that different circuits require different sequences of transformations [1]. Therefore, the increasing demand for high quality-of-result (QoR) has prompted the exploration of circuit-specific synthesis sequences to improve the synthesis quality.

Research in circuit-specific synthesis sequences has been rising in recent years. One common attempt is to utilize machine learning techniques to classify or predict the final QoR of synthesis sequences, which can help explore different sequence options [1], [2]. However, these approaches require a large dataset of pre-defined synthesis sequences for training and evaluation and usually have limited accuracy.

Recently, researchers have shown increasing interest in generating the sequence with specific optimization objectives. Several reinforcement learning (RL)-based algorithms have been proposed that treat sequence generation as a Markov decision process (MDP). These algorithms generate synthesis sequences in a step-by-step manner [3]–[7]. In addition, some studies have explored using Bayesian optimization (BO) to address this problem [8], [9]. We observe that both methods

This work is supported in part by Research Grants Council of Hong Kong SAR (No. CUHK14210723) and Shanghai Artificial Intelligence Laboratory.

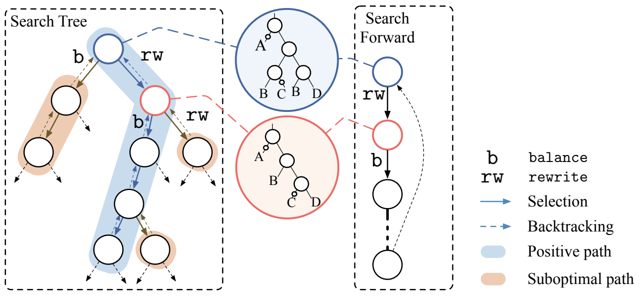

Fig. 1 Illustrations of our proposed search tree and previous one-way forward search in logic synthesis optimization.

are inefficient in exploring the solution space. Both the existing RL-based and BO-based methods perform in a "forward" process (illustrated in the right part of Fig. 1), where a sequence is first generated as a trajectory and then evaluated as a whole. They lack enough exploration, especially on the *earlier* transformation, which is critical to overall performance [6]. On the other hand, some research attempts to tackle the efficiency issue by restricting the search space with heuristics [6], [10]. Such an approach improves search efficiency and quality by allocating more exploration efforts within the reduced solution space. However, this approach overlooks the long-term effects of transformation selection and may result in non-ideal convergence to a local optimum.

Exploring the logic optimization sequences for a circuit can be fundamentally regarded as **constructing a search tree**. Each stage of the sequence is equated to a node within this tree. At such a node, positive feedback could encourage us to proceed further. However, in a suboptimal decision, the opportunity to revisit and revise the decision is equally important. This balancing act between pushing forward and going back is often a matter of managing resources and time, known as the **exploration-exploitation** trade-off. The dilemma here lies in deciding whether to keep moving forward in search of further outcomes or to retrace steps to choose a different path (as illustrated in the left part of Fig. 1). The search strategy on the search tree ultimately determines the efficiency of logic synthesis sequence generation.

In this paper, we propose to use Monte Carlo tree search (MCTS) to aid sequence generation in logic optimization. We design a framework, **AlphaSyn**, for logic sequence generation, by customizing MCTS with the domain-knowledge in logic synthesis. It leverages MCTS to balance the exploration-exploitation trade-off and hence maximizes the search efficiency of the logic optimization sequence. The contributions of this paper are summarized as follows:

- We introduce MCTS to logic synthesis sequence generation and develop a framework, AlphaSyn, for efficient exploration of logic transformation.
- We incorporate domain knowledge in logic synthesis and propose a customized MCTS algorithm. A synthesis-specific upper confidence bound for trees (SynUCT) algorithm is designed for the selection phase, balancig both the exploration-exploitation and immediate-long term trade-off. Simultaneously, a corresponding back-propagation algorithm is designed for backtracking and updating the statistics.
- We propose techniques to accelerate AlphaSyn further. Resources for sequence stage exploration are allocated by a significance-based allocator, which allows AlphaSyn to prioritize and concentrate on more crucial steps. An asynchronous parallel algorithm is proposed to expedite the exploration of MCTS within AlphaSyn.
- For standard-cell technology mapping on the ASAP 7nm library among other tasks, experimental results demonstrate that AlphaSyn surpasses SOTA FlowTune with an average 8.74% reduction in area, while exhibiting a 1.24× acceleration in runtime.

II. Background

A. Logic Optimization and Technology Mapping

Logic synthesis converts register-transfer level (RTL) circuit descriptions into gate-level netlists, optimizing design objectives like area, delay, and power. Logic optimization commonly employs a directed acyclic graph (DAG) representation, which captures the circuit logic as a logical network. And-inverter-graph (AIG) [11] is a commonly used Boolean-complete representation for logic network, where each node represents a 2-input `AND` gate and each directed edge can be marked as a `NOT` gate (inverter). The logic network plays a central role in logic optimization and technology mapping processes. Logic optimization involves transforming the AIG based on various objectives, such as minimizing the number of nodes or logic levels. In technology mapping, the logic-optimized AIG is obtained by logic minimization, and the logic network is then expressed using a set of pre-designed and pre-characterized gates from a technology library [12]. The processing of logic optimization is independent of the target implementation technology. However, the logic optimization outcomes have a direct impact on the goals of technology mapping, altering the area and delay of the resulting netlists.

ABC [13] is a synthesis and verification framework with multiple synthesis transformations that target various logic optimizations on AIG and enable technology mapping. These well-designed transformations, including `rewrite (rw)`, `rewrite -z (rwz)`, `balance (b)`, `refactor -z (rfz)`, `refactor (rf)`, and `resub -K 6 (rsk6)`, are built with various algorithms and target different objectives. We evaluate AlphaSyn on both logic optimization and technology mapping on the well-adopted ABC system.

B. Synthesis Sequence and Search Space Exploration

Circuit logic optimization typically employs a synthesis sequence, which involves iteratively applying synthesis transformations to the logic network. The effect of a transformation on a particular logic network is in general hard to predict. Consequently, evaluating a synthesis sequence often requires applying the transformations to the actual circuit, which can be time-consuming. When generating a circuit-specific synthesis sequence, a typical approach is to decide the appropriate synthesis transformation to apply at each step [3], [4]. Given n synthesis transformations to select and a total sequence length of L, we have the theoretical search space of the synthesis sequence: n^L, which is exponentially growing w.r.t the length L. The SOTA synthesis sequence generation algorithm, FlowTune, limits the theoretical search space to a *multi-set* permutation problem by enforcing all the transformations to be applied a fixed number of times to reduce search space and improve exploration efficiency [1], [6]. For example, *m-repetition* sequences are the sequences where each transformation is applied *m* times.

III. Motivation

We conduct motivational analysis and observe the following properties of the logic synthesis sequences. These observations inspire us to design the AlphaSyn algorithm.

Earlier transformation matters. As detailed discussed in FlowTune [6], the *earlier* transformations work effectively during the optimization of the logic network and dominate the synthesis sequence's performance. In the unrestricted sequence, it is more crucial to employ this insight to make full use of the resources due to the exponentially growing search space. In fact, for MCTS in AlphaSyn, we emphasize the "accumulated statistics" of the search tree, which is closely related to this observation.

Long-term effect of the transformation selection. We take the logic optimization task as an example, whose objective is to reduce the number of `and_node` in the AIG. A *greedy* strategy is to select the transformation that reduces nodes the most at each step, which is driven by immediate-return. Meanwhile, a counterpart algorithm is designed by modifying the greedy algorithm through forcing the `balance (b)` on the second index, which is not designed to reduce the node number. We build two sequences with these two algorithms and apply them to the design `bfly` from the VTR 8.0 benchmark [14]. The transformed `and_nodes` (#TNodes) that represents the `and_node` number reduced by the corresponding AIG transformations are calculated. And the final AIG `and_node` numbers (#Final) after applying these sequences are represented. As in TABLE I, the `balance (b)` in modified_greedy sequence reduced much fewer nodes (130) than the `refactor -z (rfz)` (671) in

TABLE I Results for greedy and modified greedy algorithms with design `bfly` from VTR 8.0 benchmark [14]. “#Tnodes” denotes the number of the AIG `and_node` number Transformed by the corresponding AIG transformation. “#Final” denotes the final AIG `and_node` number.

Greedy	`rw`	`rfz`	`rwz`	`rf`	`rsk6`	`rwz`	#Final
#TNodes	2083	671	319	250	172	91	25324
Modified_Greedy	`rw`	**`b`**	`rf`	`rsk6`	`rwz`	`rf`	#Final
#TNodes	2083	**130**	963	265	174	140	25155

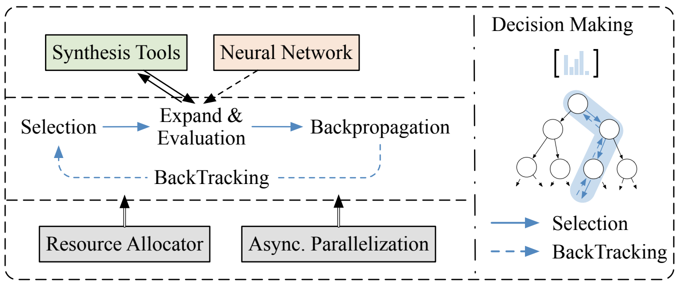

Fig. 2 The overview of AlphaSyn.

the greedy sequence. However, for the final `and_node` number, the modified_greedy algorithm can even achieve better performance. Such phenomenon can be due to the fact that `balance (b)` is useful for restructuring logic, allowing more node reduction possibilities, which is considered to be the long-term effect. Therefore, taking both immediate and long-term effects into account is necessary to select each synthesis transformation and optimize the synthesis sequence.

Based on the above analysis, we develop our domain-knowledge driven MCTS algorithm.

IV. Framework

In this section, we describe our framework AlphaSyn from an overview to the details of domain-specific MCTS, learning strategies for MCTS, and acceleration techniques.

A. Framework Overview

AlphaSyn incorporates a domain-specific MCTS to build the search tree for synthesis sequence generation. The framework overview is illustrated in Fig. 2. Within the MCTS, there are mainly three phases: *Selection*, *Expansion & Evaluation*, and *Backpropagation*. The Selection phase performs forward selection on the search tree, while the backpropagation phase performs backward statistics update. Between them, the Expansion & Evaluation phase expands new nodes and evaluates the selection results by interacting with the synthesis tools, with or without the neural network assistance. This process is iteratively implemented by backtracking to the root node and re-selecting based on previous statistics.

In the rest of this section, we presents the details of the AlphaSyn framework in enabling the efficient and effective logic synthesis sequence exploration.

- **Customized MCTS for Logic Synthesis Optimization.** Previous MCTS algorithms implemented on board games are designed based on the acquisition of reward after the entire sequence [15], [16]. However, this is inefficient for logic synthesis optimization, where the reward can be obtained immediately by interacting with the synthesis tools. We develop the customized algorithm SynUCT, which can take the immediate-long term effect into consideration combined with the exploration-exploitation trade-off. (Section IV-B)
- **Stable Learning Strategies.** Despite AlphaSyn can already achieve SOTA performance, the results can still have a bit of variation due to the exploratory search in MCTS. Therefore, several learning strategies are proposed to enhance the stability of MCTS with a neural network (PQnet), where the past observations are learned to assist present sampling. (Section IV-C)
- **Acceleration for MCTS.** Although AlphaSyn has the ability to efficiently balance the exploration-exploitation trade-off, it needs resources to execute the search algorithm and interact with the synthesis tools. We design a resource allocator and an asynchronous parallel algorithm to accelerate the MCTS, which can further reduce the runtime and maintain the performance. (Section IV-D)

B. Domain-Specific MCTS

AlphaSyn uses a customized MCTS algorithm to explore the logic synthesis sequence generation, which is widely used in RL. Given an AIG G_0 of a circuit design, assume that we need to generate a sequence of length L with synthesis transformations T_i, $i = 1, 2, 3, \ldots, L$. By conducting the synthesis transformation T_i, new AIG G_i is iteratively constructed from the previous one G_{i-1}, $G_i = T_i(G_{i-1})$, until the final AIG G_L is obtained. To generate the synthesis transformation T_i, a search tree is built when regarding the AIG G_{i-1} as the root node represents the initial state s^0_{i-1}, $s^0_{i-1} = G_{i-1}$.

The search tree is progressively assembled during the search, with each node incorporated as the search algorithm continues. With more searches executed, the tree expands and increases in size and depth. The details of each phase in MCTS are described as follows.

Selection. The selection phase is illustrated in Fig. 3. Each none-leaf node in the search tree contains edges (s, a) for all the actions a in the action space $\mathcal{A}$, where these actions are the synthesis transformations to be selected. An edge between a parent node and a child node represents an AIG transformation. From the root node state s^0 (omitting subscript for convenience), we continuously select child nodes until a leaf node $s^{\mathcal{T}}$ is reached at time-step $\mathcal{T}$. At each time-step $t < \mathcal{T}$, an action (a^t) is selected according to the child nodes statistics, which is based on our synthesis-specific upper confidence bound for trees (SynUCT) algorithm. Our proposed SynUCT algorithm is a variant of the PUCT algorithm [17], where we consider both the immediate and long-term effects. Let $s^{t,a}$ be defined as the child node with a new AIG state obtained by

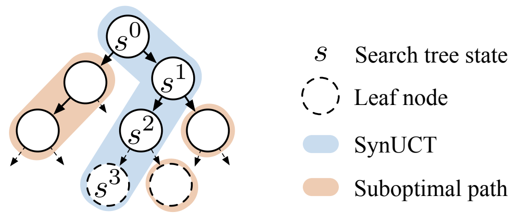

Fig. 3 The selection phase in MCTS. From the root s^0 of the search tree, transformations are selected iteratively based on SynUCT.

applying the transformation a on the AIG state s^t. Specially, $s^{t+1} = s^{t,a^t}$. The proposed SynUCT is defined as follows:

$$a^t = \arg\max_{a\in\mathcal{A}}(Q^{t,a} + R^{t,a} + U^{t,a}). \tag{1}$$

The action a^t that achieves the max $(Q{+}R{+}U)$ value will be selected. $Q^{t,a}$ represents the **long-term return** that the AIG state s applies the transformation a, which is calculated as in (6). $R^{t,a}$ represents the **immediate reward** that the AIG state s applies the transformation a, which is stored as in (3). $U^{t,a}$ is a **balance mechanism** controlling the exploitation-exploration trade-off, which is computed as follows:

$$U^{t,a} = c_{puct} \cdot P^{t,a} \cdot \frac{\sqrt{N^t}}{N^{t,a}+1}. \tag{2}$$

The c_{puct} is a constant determining the level of exploration. N^t and $N^{t,a}$ represent the number of times the current (parent) node s^t and the child $s^{t,a}$ have been visited, respectively. $P^{t,a}$ is the prior probability assigned when leaf nodes are evaluated. Particularly, we add Dirichlet noise to the prior probabilities in child nodes of the root node s^0 following the classical strategy [16]: $(1-\epsilon)P^{0,a} + \epsilon\eta$, where ϵ is a constant value and $\eta \sim Dir(\alpha)$ with parameter α. By incorporating $U^{t,a}$, SynUCT prefers the actions with low visit count, together with the impact of immediate and long-term returns. The structure of SynUCT results in a powerful form of iterated exploration on both immediate-long-term and exploitation-exploration trade-offs.

Expansion & Evaluation. When the selection stops at the leaf node $s^{\mathcal{T}}$, the new AIG is generated and the reward $R^{\mathcal{T}}$ is calculated after interacting with the synthesis tools to obtain the objective value. The objective value at time step t is denoted as O^t. For example, in the task of `and_node` minimization, O^t represents the `and_node` number for AIG state s^t. Then $R^{\mathcal{T}}$ is defined as follows:

$$R^{\mathcal{T}} = \mathrm{sgn}(O^{\mathcal{T}-1} - O^{\mathcal{T}}) \cdot \sqrt{\frac{|O^{\mathcal{T}-1} - O^{\mathcal{T}}|}{\textit{baseline}}}. \tag{3}$$

The square root $\sqrt{\cdot}$ is applied as a normalization of the objective values on different sequence steps. The *baseline* is also a normalization, computed as the mean reduction on the objective by applying a heuristic script on the initial AIG G_0. In case the heuristic script cannot give a reduction on the objective, *baseline* is set as $1/1000$ of the initial objective for G_0. A sign function sgn is set in case that the new objective $O^{\mathcal{T}}$ is larger than $O^{\mathcal{T}-1}$.

AlphaSyn can optimize synthesis with multi-objectives, which is important to explore the Pareto-optimal for trade-off synthesis objectives, such as area and delay. AlphaSyn addresses this through the integration of linear combinations for trade-off objectives:

$$R^{\mathcal{T}} = (1-\beta)\cdot R_1^{\mathcal{T}} + \beta \cdot R_2^{\mathcal{T}}, \tag{4}$$

where the reward stored in the search tree ($R^{\mathcal{T}}$) is calculated by combing the rewards of trade-off objectives, $R_1^{\mathcal{T}}$ and $R_2^{\mathcal{T}}$. A parameter, β, is employed to balance these objectives.

If the leaf node is not a terminal node, edges representing actions a in $\mathcal{A}$ are added. The prior probabilities $p^{\mathcal{T},a}$ and long-term return estimation $Q_{es}^{\mathcal{T}}$ is then assigned. For network-assisted cases, a dual-head neural network is employed to produce them. For a network-free alternative, the prior probabilities are set as the uniform distribution, and $Q_{es}^{\mathcal{T}}$ is set as 0. Finally, the new leaf nodes are initialized with initial statistics as $\{N^{\mathcal{T},a}=0, Q^{\mathcal{T},a}=0, P^{\mathcal{T},a}=p^{\mathcal{T},a}\}$.

Backpropagation. Once $R^{\mathcal{T}}$ has been calculated, the algorithm starts to backtrack and update the information of all nodes visited during the selection phase, i.e., $s^0, s^1, s^2, \ldots, s^{\mathcal{T}}$. Information is recursively updated with the order from the leaf node $s^{\mathcal{T}}$ to the root node s^0. First, the node visit count is updated:

$$N^t \longleftarrow N^t + 1. \tag{5}$$

Then, the long-term return Q is updated for $t < \mathcal{T}$:

$$Q^t \longleftarrow \lambda \cdot \max_{a\in\mathcal{A}}(Q^{t,a} + R^{t,a}), \tag{6}$$

where λ is a constant discount, controlling the balance of the immediate-long-term trade-off. $Q^{t,a}$ is represented by the estimated $Q_{es}^{t,a}$ for the nodes that have just been expanded. This algorithm chooses the max value of the child nodes' $(Q+R)$. Hence, the long-term value Q of a node is computed by aggregating the discounted rewards and discounted estimated long-term returns of its successors. On the other hand, the estimated Q_{es} can be relatively inaccurate for the deeper nodes as they are less frequently visited. The discount factor λ serves as a utility to reduce the impact of those nodes and stabilize the search process.

The design of the long-term return Q is based on our motivational analysis shown in Section III. The accumulated Q prioritizes the exploration in the shallow levels, which corresponds to the early transformations in a sequence. On the other hand, our MCTS and long-term return estimation mechanism also encourage deeper exploitation in promising directions with high potential. The prior probabilities and long-term return estimation provide guidance on the search direction in awareness of the historical experience.

Decision Making. The decision making phase is demonstrated in Fig. 4. After a predetermined number of searches, the statistics of the child nodes of the root node s^0 are

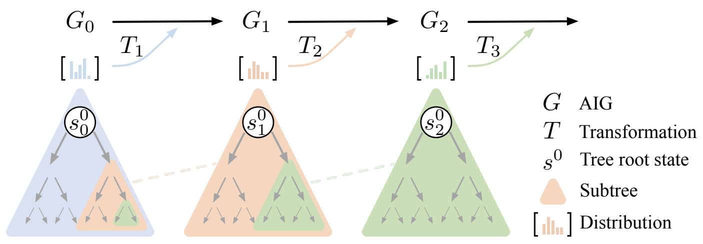

Fig. 4 Decision making in AlphaSyn. After repeatedly conducting the search algorithm, the decision-making process is done with the accumulated statistics from the search tree.

summarized. Unlike previous classical MCTS works [15], [16], AlphaSyn does not utilize the visit counts distribution of the child nodes. Instead, we directly select the action with the max $(Q + R)$ as the generation result of synthesis transformation T_i:

$$T_i = \arg\max_{a \in \mathcal{A}} (Q^{0,a} + R^{0,a}). \tag{7}$$

The reason is that, in the situation of logic synthesis, the better or worse of the synthesis transformation can be explicitly represented by the combination of immediate and long-term returns. Therefore there is no need to employ the indirect representation from visit counts.

Upon determining the final synthesis transformation, the search tree can be reused for subsequent generations by designating the child node corresponding to the determined T_i as the new root node. The sub-tree beneath this child node is preserved, along with all associated statistics, while the rest of the tree is discarded.

C. Learning Strategies for MCTS

We propose neural network-based learning strategies to enhance the stability of AlphaSyn and learn from past generations. Our approach includes introducing customized methods for network data collection *self-syn*, refining network architecture, and implementing relevant techniques to improve the network's execution efficiency.

Self-Syn. We refer to *self-syn* as a process to conduct multiple rounds of synthesis sequence generation to collect training data. Self-syn is implemented differently from optimization-driven synthesis. We aim for a more exploratory approach to collect data for richer and more comprehensive samples. First, self-syn updates the long-term return Q by taking the mean of the child nodes' return $(Q + R)$ rather than choosing the maximum value. Second, for the decision making part, self-syn picks an action proportional to the exponential of return $(Q + R)$ instead of the max one:

$$\pi^{0,a} = \frac{(Q^{0,a} + R^{0,a})^{1/\kappa}}{\sum_{b \in \mathcal{A}} (Q^{0,b} + R^{0,b})^{1/\kappa}}, \tag{8}$$

where κ is a temperature parameter. Increasing κ can flatten the distribution for more even exploration. For self-syn, we schedule κ in a gradually decreasing manner: $\kappa_i = \kappa_{\text{init}} \cdot (1 - \frac{i-1}{L-1}) + \frac{i-1}{L-1}$, where κ_{init} is the initial temperature.

Network I/O and Training. During the selection phase of MCTS, a GNN-based dual-head neural network, PQnet (f_θ), is employed to assist the exploration when encountering leaf nodes $s^{\mathcal{T}}$. The input to PQnet includes the following: The search tree state $s_i^{\mathcal{T}}$ (an AIG), its last transformation ($a_i^{\mathcal{T}-1}$ in $s_i^{\mathcal{T}} = s_i^{(\mathcal{T}-1),a_i^{\mathcal{T}-1}}$;), and the corresponding position index ($(\mathcal{T}-1)+i$ for $a_i^{\mathcal{T}-1}$). After processing, PQnet outputs the prior probability distribution $\boldsymbol{p}^{\mathcal{T}}$ and the estimated long-term return $Q_{es}^{\mathcal{T}}$:

$$(\boldsymbol{p}^{\mathcal{T}}, Q_{es}^{\mathcal{T}}) = f_\theta(s_i^{\mathcal{T}},\ a_i^{\mathcal{T}-1},\ (\mathcal{T}-1)+i). \tag{9}$$

As a general example, for the initial state s_i^0, the inputs are given as G_i, T_i, and i, respectively. For the initial design AIG G_0, we assign 0 as the last action and the position index. The two index scalars are embedded in high-dimensional space.

PQnet is trained and refined in the RL process. Training is based on supervision from root node statistics on the search tree, summarized for each single transformation generation. For each constructed search tree in self-syn, the distribution $\boldsymbol{\pi}^0$ in (8) and the long-term return value Q^0 of the root node are collected. The first objective is to minimize the error of policy prediction between the distributions $\boldsymbol{p}^0$ and $\boldsymbol{\pi}^0$. The second objective is to minimize the error of value prediction between Q_{es}^0 and Q^0. The overall loss is:

$$L_{\text{total}} = L_{\text{CE}}(\boldsymbol{p}^0, \boldsymbol{\pi}^0) + L_{\text{MSE}}(Q_{es}^0, Q^0), \tag{10}$$

where L_{CE} denotes the cross-entropy loss and L_{MSE} denotes the mean-squared error. AlphaSyn continually trains PQnet by collecting training samples with the latest checkpoint, updated with every fixed number of self-syn steps.

Network Architecture. We design the PQnet based on SAGEConv [18] and the self-attention pooling SAGPool [19]. As demonstrated in Fig. 5(a), the features of AIG are extracted by self-attention blocks and residual blocks, whose structures are shown in Fig. 5(b) and Fig. 5(c). The output is generated by the P-head and Q-head, which have the same structure, consisting of a residual block, a convolutional layer, and a multi-layer linear perceptron (MLP). The output dimension of the MLP is the number of actions for the probability output ($\boldsymbol{p}$) and 1 for the return output (Q_{es}).

Data Reuse. As PQnet keeps updating, the statistics summarized on each search tree root may also change, leading to conflicting labels and biased data. To address the issue, we propose a tree-merging technique. Specifically, we maintain a set of *main trees* from the beginning, which stores all the data during training. When collected, new samples are merged into the corresponding *main tree* with the same source state. With tree-merging, AlphaSyn can get rid of the influence of biased data.

Stop Index. The more transformations are applied, the smaller the difference between AIGs. It is not a good choice to use them all as training data. Therefore, we stop running PQnet during sequence generation when there are no conspicuous changes on AIGs, which is controlled by an

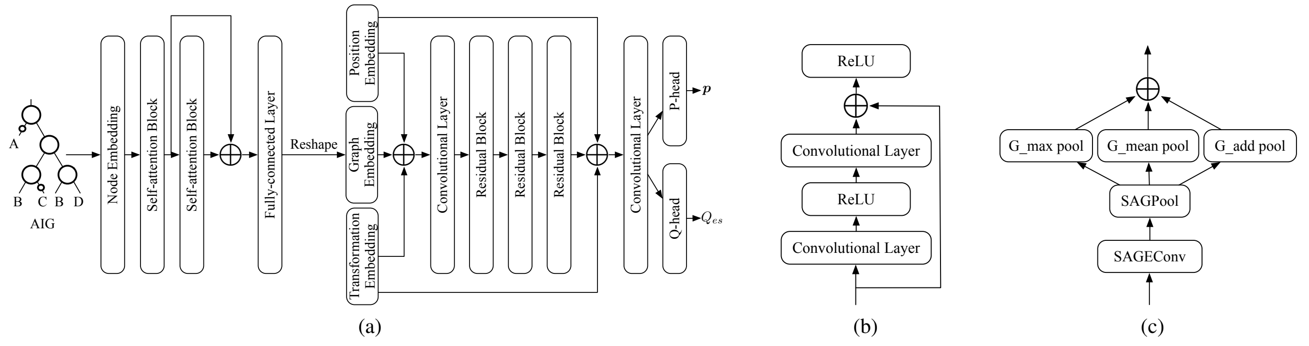

Fig. 5 (a) The overview of PQnet; (b) Residual block; and (c) Self-attention block. Two self-attention blocks extract features of AIG. "G_max", "G_mean", and "G_add" denote the global_max, global_mean, and global_add pooling, respectively.

integer hyperparameter $stop\_index$. Doing this strengthens the significant part during sequence generation, and the runtime can be largely saved. We believe this is a natural tradeoff between performance and runtime cost.

D. Acceleration Techniques for AlphaSyn

AlphaSyn utilizes resource allocation and asynchronous parallel mechanisms to accelerate the MCTS, effectively reducing runtime while ensuring maintained performance.

Resource Allocator. As optimization converges, the importance of synthesis transformation is also diminishing. Simultaneously, MCTS is, in essence, a high-computational process; a better allocation of resources can significantly improve the efficiency of AlphaSyn. Therefore, a simple but effective resource allocator is designed. The allocator works in two parts. For the earlier generations, the search times are gradually decreased as the index increase, which is controlled by: $n^i_{sche} = n_{init} - D \cdot (i-1)$, where n^i_{sche} is the number of searches for generating T_i, n_{init} is an initial number, and D is the decrement value. For the rest of the generation, we determine a fixed number of searches for each step, i.e., n_{base}. These two parts are separated by a split index I_{split}. Typically, we assign a much higher value for n_{init} than n_{base}.

Asynchronous Parallelization. To further improve the exploration efficiency, we design a novel asynchronous parallel algorithm with multi-threads for the search process. The interaction with synthesis tools in the search process is the runtime bottleneck for AlphaSyn. We develop a conditional path-blocking-releasing algorithm in conjunction with thread locking to address this issue. For each node s in the search tree, we define a special value B (initialized to 0) to represent whether or not the node should be blocked. When a leaf node s^l is accessed during the selection process, its B^l is assigned an extremely small negative value M. Additionally, the ancestors of the leaf node, $s^{l-1}, s^{l-2}, \dots, s^1$, are recursively examined to determine if the number of their blocked child nodes exceeds a threshold value B_{thre}. If the threshold is exceeded, the ancestor's B value is also assigned M. The role of B is to incorporate it into SynUCT as Par-SynUCT:

$$a^t_{par} = \arg\max_{a\in\mathcal{A}}(Q^{t,a} + R^{t,a} + U^{t,a} + B^{t,a}). \tag{11}$$

By doing this, the blocked nodes will never be selected by Par-SynUCT due to the extremely small value of $B^{t,a}$. Therefore, the leaf nodes are being accessed, and their ancestors in one thread are all conditionally ignored by other threads. Finally, at the end of the backpropagation phase, the leaf node is released by assigning B^l back to 0; its ancestors are also recursively released when the threshold condition is unmatched.

V. Experiments

We implement AlphaSyn with Python and the training of neural network with Pytorch and Pytorch Geometric. We use the open-source tool Yosys [20] to perform logic synthesis in conjunction with ABC [13]. The prepossessing of AIG for GNN is developed with the help of a python interface of ABC [4]. The experiments are conducted on a machine with 128 core Intel® Xeon® Platinum 8358P CPU @ 2.60GHz and an NVIDIA A100-SXM4-80GB (Ampere architecture, SM80) graphics card with CUDA Driver 11.4.

Due to the high scalability of AlphaSyn, it can be compared with previous works on different benchmarks and settings. The experiments are organized according to the optimization objectives. We use the heuristic script `resyn2` for the *baseline* acquirement in (3). The results are presented for both network-free ("w/o nn") and network-assisted ("w/ nn") versions. For network-assisted versions, the network is trained for 10 hours to 24 hours, which depends on the circuit size. We use "rt" to denote runtime of optimization. For AlphaSyn, the runtime refers to the inference time of sequence generation, which includes the search time in MCTS. Bar notation $\bar{\cdot}$ and hat notation $\hat{\cdot}$ are used to denote average results and best results, respectively.

A. Logic Optimization

For logic optimization, the objective is to minimize the number of AIG `and_node`, which is denoted by "#N". We present the best ($\hat{\#N}$) and average ($\bar{\#N}$) results for AlphaSyn.

The performance of AlphaSyn is compared with the SOTA, FlowTune [6], on the same selected VTR benchmarks [14]. The action space is $\mathcal{A} = \{$`balance`, `rewrite`, `rewrite -z`, `refactor`, `refactor -z`, `resub -K 6`$\}$. We set

TABLE II Comparison with FlowTune [6] for logic optimization.

Design	Flowtune [6]		AlphaSyn w/o nn		AlphaSyn w/ nn		
	#N	rt (s)	$\#\bar{N}$	$\bar{rt}$ (s)	$\#\hat{N}$	$\#\bar{N}$	$\bar{rt}$ (s)
bfly	22740	495.81	22619.9	**223.60**	**22381**	**22582.7**	537.39
dscg	22258	482.35	22225.2	**210.20**	**22000**	**22155.9**	557.09
fir	21807	488.67	21611.6	**211.40**	**21544**	**21591.5**	543.77
ode	13038	260.16	12935.9	**112.98**	**12736**	**12768.7**	472.55
or1200	10316	118.80	10266.9	**74.16**	**10194**	**10195.2**	420.02
syn2	23633	504.19	23547.1	**242.00**	**23233**	**23535.7**	587.01
Average	18965.3	391.66	18867.8	**179.06**	**18689.5**	**18805.0**	519.64
Ratio	1.000	1.000	0.995	**0.457**	**0.985**	**0.992**	1.327

TABLE III Comparison with MLCAD'20 [4] and ICCAD'21 [5] for logic optimization.

Design	[4]	[5]	AlphaSyn w/o nn		AlphaSyn w/ nn		
	#N	#N	$\#\bar{N}$	$\bar{rt}$ (s)	$\#\hat{N}$	$\#\bar{N}$	$\bar{rt}$ (s)
C1355	386.2	**386**	**386**	19.24	**386**	**386**	360.93
C6288	**1870**	**1870**	**1870**	24.35	**1870**	**1870**	454.89
C5315	1337.4	1315	1291.2	20.87	**1287**	**1289.4**	450.44
dalu	1039.8	1085	1008.4	21.22	**1007**	**1008.4**	475.94
k2	1128.4	1137	1045.9	22.66	**1035**	**1041.3**	451.69
mainpla	3438.4	3461	3406.3	24.81	**3386**	**3390.2**	462.66
apex1	1921.6	1885	1892	22.89	**1881**	**1883**	453.62
bc0	819.4	831	803.2	23.14	**795**	**798**	464.43
Average	1492.7	1496.3	1462.9	22.40	**1455.88**	**1458.3**	446.83
Ratio	1.000	1.002	0.980	-	**0.975**	**0.977**	-

TABLE IV Comparison with FlowTune [6] for technology mapping on Nangate 45nm library.

Design	Flowtune [6]		AlphaSyn w/o nn		AlphaSyn w/ nn		
	Area (μm^2)	rt (s)	$\overline{\text{Area}}$ (μm^2)	$\bar{rt}$ (s)	$\widehat{\text{Area}}$ (μm^2)	$\overline{\text{Area}}$ (μm^2)	$\bar{rt}$ (s)
bfly	16881.8	969.95	16089.0	**450.78**	**15493**	**15959.2**	830.20
dscg	16393.2	913.20	16142.2	**411.80**	**15743**	**16097.1**	958.01
fir	16323.5	922.73	15876.9	**431.70**	**15336**	**15724.4**	896.26
ode	9302.5	479.77	9138.4	**272.59**	**9101**	**9151.2**	710.97
or1200	6731.4	228.24	6756.6	**153.92**	**6689.6**	**6729.1**	531.53
syn2	17246.4	880.14	16410.6	**490.37**	**16051.5**	**16078.1**	943.32
Average	13813.1	732.34	13402.3	**368.53**	**13069.0**	**13289.9**	811.72
Ratio	1.000	1.000	0.970	**0.503**	**0.946**	**0.962**	1.108

TABLE V Comparison with FlowTune [6] for technology mapping on ASAP 7nm library.

Design	Flowtune [6]		AlphaSyn w/o nn		AlphaSyn w/ nn		
	Area (μm^2)	rt (s)	$\overline{\text{Area}}$ (μm^2)	$\bar{rt}$ (s)	$\widehat{\text{Area}}$ (μm^2)	$\overline{\text{Area}}$ (μm^2)	$\bar{rt}$ (s)
bfly	16934.2	2111.12	15586.7	**1396.17**	**14837.7**	**15059.7**	1520.04
dscg	15722.3	1865.30	15537.2	**1261.43**	**15031**	**15164**	1329.71
fir	16109	1784.02	16225.8	**1060.90**	**15301**	**15471**	1294.33
ode	9193.4	979.36	8319.8	**763.16**	**8123**	**8257.3**	919.07
or1200	**4107.6**	442.80	4241.9	**584.95**	4191.3	4201.1	719.72
syn2	18501.4	1812.36	15728.9	**1254.69**	**15217.3**	**15370.2**	1480.61
Average	13428	1499.16	12606.7	**1053.55**	**12116.9**	**12253.9**	1210.58
Ratio	1.000	1.000	0.939	**0.703**	**0.902**	**0.913**	0.808

the "stages:iterations" (s:m) number as 2 : 30 with *2-repetition* for all the experiments with FlowTune, which is reported to have good performance. Then the total sequence length L becomes 24. Following FlowTune implementation, a subsquence `ifraig; dch -f` is added at the end of each stage. AlphaSyn adds this subsequence at sequence index 12 and 24. The results are represented in TABLE II. Compared with FlowTune, AlphaSyn has 0.51%, 1.50% and 0.85% performance gain on logic optimization for average "w/o nn", best "w/ nn", and average "w/ nn", respectively. Meanwhile, nn-free AlphaSyn has a 2.19× runtime speedup, while the nn-assisted version maintains a reasonable runtime (around 30% overhead). The results demonstrate the superior performance and efficiency of AlphaSyn over previous approaches. Besides, the neural network can indeed enhance the stability of AlphaSyn.

We also compare our results with MLCAD'20 [4] and ICCAD'21 [5] for logic optimization, directly drawing from the same benchmarks and data presented in their original papers. For ICCAD'21, the results in "ENV1" is compared. The action space is $\mathcal{A} = \{$`balance`, `rewrite`, `rewrite -z`, `refactor`, `refactor -z`$\}$, and the sequence length L is 20. The results are represented in TABLE III. AlphaSyn achieves an average improvement of 1.99% for nn-free and 2.30% for nn-assisted versions, compared with MLCAD'20. The improvement is up to 2.46% with neural network assistance.

B. Standard-cell Technology Mapping

For standard-cell technology mapping, the objective is to optimize the area after technology mapping from the standard-cell library. "Area" denotes the circuit area after mapping. We present the best ($\widehat{\text{Area}}$) and average ($\overline{\text{Area}}$) results for AlphaSyn.

In comparison to FlowTune [6], our experiment follows a similar setup as described in Section V-A. We conduct experiments on two different standard-cell libraries: Nangate 45nm and ASAP 7nm. The area information is collected using the `map` command in ABC. Experimental results are illustrated in TABLE IV and TABLE V for Nangate 45nm and ASAP 7nm, respectively. Regarding the Nangate 45nm library, AlphaSyn achieves performance improvements of 2.97%, 5.39%, and 3.79% for average "w/o nn", best "w/ nn", and average "w/ nn". Meanwhile, the nn-free version of AlphaSyn demonstrates a 1.99× faster runtime, while the nn-assisted version maintains a comparable runtime (10% overhead). AlphaSyn also shows performance improvement across all benchmarks for the ASAP 7nm library, with the mean "w/o nn" case improving by 6.12%, the best "w/ nn" case by 9.76%, and the mean "w/ nn" case by 8.74%. Additionally, AlphaSyn exhibits runtime speedup of more than 1.24×. These results indicate that AlphaSyn outperforms previous SOTA methods in terms of performance and efficiency for standard-cell technology mapping on different standard-cell libraries.

C. FPGA Technology Mapping

For FPGA technology mapping, the objective is to minimize the total count of LUT-6. We denote LUT-6 count as "LUTs" after mapping. The best and average results for AlphaSyn are represented as $\widehat{\text{LUT}}$s and $\overline{\text{LUT}}$s, respectively.

TABLE VI Comparison with FlowTune [6] for FPGA technology mapping on LUT-6.

Design	Flowtune [6]		AlphaSyn w/o nn		AlphaSyn w/ nn		
	LUTs	rt (s)	$\bar{\text{LUTs}}$	$\bar{\text{rt}}$ (s)	$\hat{\text{LUTs}}$	$\bar{\text{LUTs}}$	$\bar{\text{rt}}$ (s)
bfly	8187	1511.40	7974.6	**945.39**	**7889**	**7949.9**	1148.27
dscg	8191	1481.99	8018.9	**1005.74**	**7965**	**7972.9**	1065.17
fir	7958	1422.34	7815.6	**952.29**	**7728**	**7778.1**	1036.73
ode	5022	708.07	5010.7	**593.29**	**4973**	**4995.2**	830.44
or1200	2751	401.295	2725.2	**300.62**	**2714**	**2720.6**	564.46
syn2	8316	1510.25	8215.4	**968.38**	**8172**	**8189.7**	1142.71
Average	6737.5	1172.56	6626.7	**794.27**	**6573.5**	**6601.1**	964.63
Ratio	1.000	1.000	0.984	**0.677**	**0.976**	**0.980**	0.823

The setting for AlphaSyn is almost the same as previous experiments except for the following: In line with FlowTune's implementation, the subsequence `ifraig;scorr;dc2;strash;dch -f` is appended to each stage, while AlphaSyn inserts this subsequence at sequence indexes 12 and 24. The LUT information is gathered by executing the ABC command sequence `if -K 6;mfs2;lutpack -S 1`. Experimental results, presented in TABLE VI, demonstrate that AlphaSyn outperforms FlowTune across all benchmarks, achieving performance gains of 1.64%, 2.43%, and 2.02% in FPGA technology mapping for mean "w/o nn", best "w/ nn", and mean "w/ nn". Additionally, AlphaSyn achieves speedups of 1.48$\times$ and 1.22$\times$ for nn-free and nn-assisted versions, respectively. These results highlight the superior performance and efficiency of AlphaSyn compared with FlowTune for the LUT mapping task. Specifically, integrating neural networks greatly enhances the stability of AlphaSyn and yields superior optimization results.

We also compare with DRiLLS [3] and ASPDAC'23 on EPFL benchmarks [21] for FPGA mapping. Their results are also collected from the "Last10" in the ASPDAC'23 paper, where for ASPDAC'23 the "RL-PPO-PRuned" (the best) version is used. The action space is $\mathcal{A}$ = {`balance`, `rewrite`, `rewrite -z`, `resub`, `resub -z`, `refactor`, `refactor -z`}, and the sequence length L is 25. The LUT information is collected by conducting the command `if -a -K 6`. The experimental results are demonstrated in TABLE VII. On average, AlphaSyn demonstrates a 6.31% improvement without nn and a 6.55% improvement with nn. The improvement is up to a remarkable 10.74%.

D. Multi-objective Synthesis

We conduct experiments on the `bfly` circuit design with the setting of FlowTune [6] on logic optimization. We first randomly generate 100000 sequences, which is reasonable and comparatively large, to represent the entire search space. Based on Equation (4), we adjust the parameter β to optimize with different weights between the number of AIG `and_node` and `level`, in order to construct the Pareto front. As illustrated in Fig. 6, the results of AlphaSyn ("w/o nn") form a Pareto front that can encompass all the random search outcomes as well as the result of FlowTune. We also specially mark the result of AlphaSyn on `and_node` minimization, which is the fair comparison with FlowTune. The outcome amply demonstrates our algorithm's superiority in multi-objective optimization.

TABLE VII Comparison with DRiLLS [3] and ASP-DAC'23 [7] for FPGA technology mapping.

Design	[3]	[7]	AlphaSyn w/o nn		AlphaSyn w/ nn		
	LUTs	LUTs	$\bar{\text{LUTs}}$	$\bar{\text{rt}}$ (s)	$\hat{\text{LUTs}}$	$\bar{\text{LUTs}}$	$\bar{\text{rt}}$ (s)
max	694	687.8	680.5	74.59	**674**	**680**	342.56
adder	**244**	**244**	**244**	62.74	**244**	**244**	368.74
cavlc	112.2	111.3	106.8	53.14	**106**	**106**	321.12
ctrl	**28**	**28**	**28**	38.81	**28**	**28**	341
int2float	42.6	42.3	39.2	56.62	**39**	**39**	332.82
router	70.1	69.5	65.6	24.59	**65**	**65**	320.43
priority	133.4	142.9	135.6	59.15	**131**	135	350.11
i2c	292.1	289.32	280.6	47.78	**272**	**280**	373.46
sin	1441.5	1438	1439.7	91.02	**1435**	**1438**	406.19
square	3889.4	3889	3877	166.27	**3875**	**3877**	523.26
sqrt	4708	4685.3	**4415**	269.59	**4415**	**4415**	589.93
log2	7583.6	7580.1	**7580**	365.77	**7580**	**7580**	706.96
multiplier	5678	5672	5687.5	245.75	**5670.5**	**5672**	620.53
voter	1834.7	1678.1	1538.8	111.44	**1534**	**1537.4**	470.64
div	7944.4	7807.1	6685.3	244.04	**5088.4**	**6650.1**	712.57
mem_ctrl	10527.6	10309.7	9567.7	71.63	**9211.5**	**9513.2**	659.67
Average	2826.5	2792.2	2648.2	123.93	**2523.0**	**2641.2**	464.99
Ratio	1.000	0.988	0.937	-	**0.893**	**0.934**	-

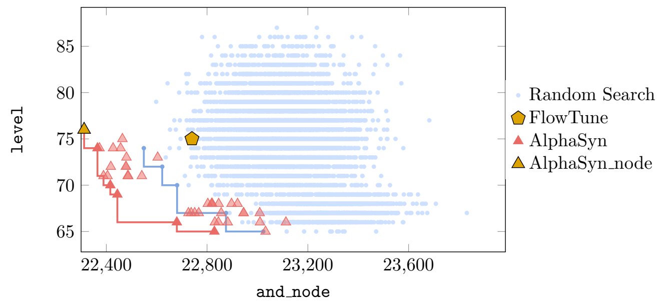

Fig. 6 Multi-objective synthesis with AlphaSyn. By tuning the weight between `and_node` and `level`, a Pareto front has been formed by AlphaSyn's results. The result on objective of pure `and_node` minimization is specially marked ("AlphaSyn_node") for fair comparison with FlowTune.

VI. Conclusion

In summary, this paper introduces a novel framework, AlphaSyn, for logic synthesis recipe generation to improve circuit design QoR. Our domain-specific Monte Carlo tree search (MCTS) algorithm improves exploration efficiency while optimizing sampling point utilization. Key contributions include customized MCTS for logic synthesis optimization, stable learning strategies, and some acceleration techniques. Our experimental results show that AlphaSyn outperforms state-of-the-art methods in various objectives while reducing runtime. Future work could explore additional domain-specific adaptations or incorporate more powerful techniques to further improve the framework's performance and applicability.

References

[1] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *ACM/IEEE Design Automation Conference (DAC)*, 2018.

[2] A. B. Chowdhury, B. Tan, R. Carey, T. Jain, R. Karri, and S. Garg, "Bulls-Eye: Active few-shot learning guided logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[3] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2020.

[4] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*, 2020.

[5] Y. V. Peruvemba, S. Rai, K. Ahuja, and A. Kumar, "Rl-guided runtime-constrained heuristic exploration for logic synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[6] W. L. Neto, Y. Li, P.-E. Gaillardon, and C. Yu, "FlowTune: End-to-end automatic logic optimization exploration via domain-specific multi-armed bandit," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[7] G. Zhou and J. H. Anderson, "Area-driven FPGA logic synthesis using reinforcement learning," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2023.

[8] A. Grosnit, C. Malherbe, R. Tutunov, X. Wan, J. Wang, and H. B. Ammar, "Boils: Bayesian optimisation for logic synthesis," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2022.

[9] C. Feng, W. Lyu, Z. Chen, J. Ye, M. Yuan, and J. Hao, "Batch sequential black-box optimization with embedding alignment cells for logic synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022.

[10] X. Li, L. Chen, F. Yang, M. Yuan, H. Yan, and Y. Wan, "HIMap: a heuristic and iterative logic synthesis approach," in *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[11] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 21, no. 12, pp. 1377–1394, 2002.

[12] J.-H. R. Jiang and S. Devadas, "Logic synthesis in a nutshell," in *Electronic Design Automation*. Elsevier, 2009, pp. 299–404.

[13] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer-Aided Verification (CAV)*, 2010.

[14] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[15] 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.

[16] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[17] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[18] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[19] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International Conference on Machine Learning (ICML)*, 2019.

[20] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[21] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IEEE/ACM International Workshop on Logic Synthesis*, 2015.