CBTune: Contextual Bandit Tuning for Logic Synthesis

Fangzhou Liu[1], Zehua Pei[1], Ziyang Yu[1], Haisheng Zheng[2], Zhuolun He[1,2], Tinghuan Chen[3], Bei Yu[1]
[1]The Chinese University of Hong Kong [2]Shanghai Artificial Intelligence Laboratory
[3]The Chinese University of Hong Kong, Shenzhen

***Abstract*—Logic synthesis pre-optimization involves applying a sequence of transformations called synthesis flow to reduce the circuit's Boolean logic graph, like AIG. However, the challenge lies in selecting and arranging these transformations due to the exponentially expanding solution space. In this work, we propose CBTune, a novel online learning framework that utilizes a contextual bandit algorithm to explore the solution space and generate synthesis flows efficiently. We develop the Syn-LinUCB algorithm as the agent, which incorporates circuit characteristics and leverages long-term payoffs to guide decision-making, thus effectively preventing getting trapped in local optima. Experimental results show that our framework achieves the optimal synthesis flow with a lower time cost, substantially reducing the number of AIG nodes and 6-LUTs compared to SOTA approaches.**

I. Introduction

Logic synthesis converts high-level circuit descriptions into gate-level netlists via translation, mapping, and optimizations. Pre-optimization, also known as technology-independent optimization, involves a time-consuming process of applying various logic equivalence rules to compact circuit's Boolean logic. And-Inverter Graphs (AIGs) serve as a common representation for logic graphs in the synthesis tool ABC [1]. Applying a synthesis flow with multiple transformations to the AIG leads to reduced logic nodes and depth, thus indirectly improving the circuits' Quality of Results (QoR).

Developing an efficient synthesis flow proves to be an intricate task. Yu *et al*. [2] indicate that different synthesis flows yield diverse optimization results for AIG. However, due to sequence length and transformation choices, exploring the optimal synthesis flow faces an exponential solution space, making manual search impractical. Additionally, they stress that an optimal flow for one design cannot be transferred for similar gains in others. Hence, although ABC provides heuristic synthesis flows like *resyn2* and *compress2*, their fixed order of operations hinders efficient logic optimization.

Machine learning is widely used in pre-optimization to accelerate design convergence and minimize manual supervision. For one thing, it involves modeling the circuit's structure and correlating sequential features with specific optimization objectives, which facilitates rapid and accurate metric estimation for synthesis flows [2]–[4]. For another, reinforcement learning is employed to actively generate synthesis flows and expedite solution space exploration. DRiLLS [5] introduced an intelligent framework that allows the A2C agent to select optimal transformations flexibly. Zhu *et al*. [6] employed GNNs to capture AIG topology and combined it with historical decisions to enrich state information for improved decision-making.

However, these methods using pre-trained models and complex networks have certain limitations. Firstly, constructing datasets and training network models for large-scale circuit designs is time-intensive and resource-demanding. Secondly, model transferability is limited due to reliance on dataset-specific objectives [2], [5]. OpenABC-D [7] revealed that the permutation similarity between optimal synthesis flows for different circuits is under 30%, underscoring the design-specific nature of synthesis flow generation and impeding model transferability. Lastly, neural network frameworks like PyTorch often introduce notable runtime overhead when communicating with C++/C-based logic synthesis tools during system integration.

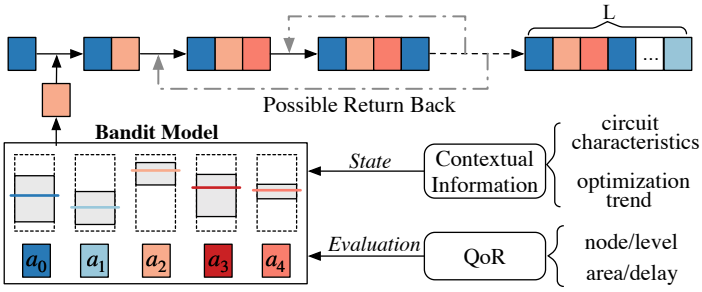


Fig. 1 Illustration of our proposed contextual bandit-based approach for efficient synthesis flow generation.

Recently, the lightweight Multi-Arm Bandit (MAB) model has demonstrated its effectiveness in identifying the optimal synthesis flow in FlowTune [8]. The core idea revolves around achieving a balance between exploring and exploiting arms through multiple trials to maximize overall payoffs. However, as a non-contextual MAB method, FlowTune's policy updates rely solely on real-time synthesis results from each arm's sampling tests, disregarding vital domain-specific arm features like optimization trends and current AIG characteristics. Furthermore, it makes decisions on a sequence-by-sequence basis, overlooking permutations within each sequence, which comes at the cost of final performance. In this paper, we present an online learning framework based on the contextual bandit model, customized for efficient synthesis flow generation, as illustrated in Fig. 1. We create separate bandit models for the selection of each transformation and try to gather domain-specific insights to guide the agent's decisions. As each model operates independently, eliminating the need for trial-and-error learning from prior model experiences, we enable local retractions and introduce a novel "return-back" mechanism that differs from traditional RL approaches in dealing with sequential decision-making tasks. Our main contributions are summarized as follows:

- Our proposed framework, CBTune, tailors the contextual bandit algorithm to facilitate efficient transformation se-

lection through iterative model tuning.

- We implement the Syn-LinUCB algorithm as the decision agent and establish a context generator for informed decision-making in the bandit model.
- We present a novel "return-back" mechanism that revisits decisions to avoid local optima, distinguishing it from typical RL scenarios.
- Experimental results show that our framework outperforms SOTA approaches within the same action space.

II. Preliminaries

A. Boolean Logic Optimization

Boolean logic optimization employs techniques like logic sharing and reusing to minimize Boolean networks. The And-Inverter Graph (AIG) is an efficient Boolean network, decomposing the circuit's logic into two-input nodes (AND function) and dotted edges (NOT function). In ABC [1], AIG-based transformations such as *rewrite* and *refactor* are widely used to achieve rapid logic reduction through graph representations. By iteratively traversing the AIG, these transformations identify appropriate nodes for optimization and update the AIG accordingly. Typically, AIG node count and depth serve as essential performance metrics, where reducing logic nodes decreases circuit size and lowering depth enhances circuit speed.

A synthesis flow strategically arranges transformations to achieve optimized performance. Let $\mathcal{A} = \{a_0, a_1, a_2, ..., a_n\}$ represents a set of n candidate transformations, and $\mathbb{F}$ represents a synthesis flow comprising permutations of a_k selected from $\mathcal{A}$. Consequently, when seeking to generate a $\mathbb{F}$ of length L, the solution space for making selections is of size n^L. Given the time overhead of evaluating each selected transformation's optimization result, efficient exploration becomes crucial.

B. Bandit Problem

The Multi-Arm Bandit (MAB) problem involves selecting arms within a limited number of trials, striking a balance between "exploitation" and "exploration" to maximize overall payoff. One prominent MAB algorithm, Upper Confidence Bound (UCB) [9], ingeniously devises a trade-off strategy by extending a fluctuation range Δ. UCB estimates each arm's observed payoff p' based on historical trial data and iteratively approximates its true payoff p, maintaining the relationship: $p' - \Delta \leq p \leq p' + \Delta$. As the trial progresses, the fluctuation range Δ gradually decreases to zero, leading to $p' \to p$. The UCB_i (i.e., p') for the i-th arm in iteration t is given by:

$$\text{UCB}_i = \hat{x}_i(t) + \Delta = \hat{x}_i(t) + \sqrt{\frac{2\ln(t)}{T_{i,t}}}. \tag{1}$$

Here, $\hat{x}_i(t)$ estimates the payoff for selecting arm i up to iteration t by averaging historical results. The latter term captures the fluctuation range, with $T_{i,t}$ tracking the number of times arm i has been selected until iteration t. This strategy efficiently balances the exploration of uncertain arms, even if their observed payoffs are lower, with the exploitation of known arms by allocating more trials to arms with higher upper bound and fewer trials to those with lower upper bound.

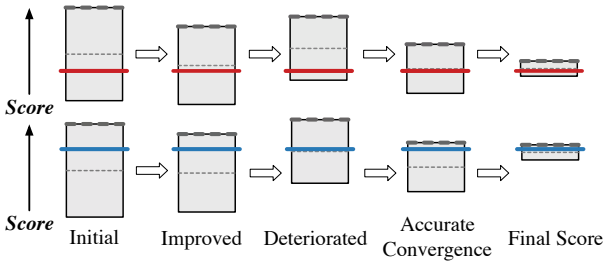


Fig. 2 Score iterations for each arm in bandit model.

C. Motivation

We seek to formulate a domain-specific bandit model for any single transformation's decision-making in the synthesis flow and assess their feasibility. In this model, each transformation in the candidate set $\mathcal{A}$ serves as an "arm", initially assigned uniform UCB scores. The model iterates to update each arm's score and differentiates their performance. Simultaneously, it selects the arm with the highest score in each iteration, adjusting its scoring parameters and narrowing the confidence interval for improved reliability. Ultimately, each arm's score converges to its true payoff, with the highest-scoring arm indicating optimal optimization performance.

In Fig. 2, we visualize how arm scores are iteratively updated in the bandit model. The red and blue lines denote each arm's unknown true payoff. This uncertainty arises as we can only observe the immediate synthesis results from applying a single transformation a of each arm, without foreseeing its long-term payoff in the total flow, which will be elaborated upon in Section III-B. Each arm's score corresponds to the upper bound of its gray confidence interval, marked by a dashed line. As iterations proceed, the confidence interval gradually narrows, and the gray dashed line aligns with red and blue lines due to continuously updated decision parameters. After sufficient iterations for convergence, informed choices can be made based on each arm's score.

III. Algorithms

A. CBTune Framework

Our proposed framework, CBTune, decomposes the synthesis flow of length L into a series of sequential steps, each employing a domain-specific bandit model, as depicted in Fig. 3. Unlike Flowtune [8], which relies on segmented sequences for decision-making, our framework generates the synthesis flow step by step, guided by analyzing each transformation's effectiveness within the sequence. In Fig. 4(a), we introduce random disturbance to the transformation at each step of *resyn2* by substituting them with alternatives from the candidate set $\mathcal{A}$. The results reveal that modifying a single transformation at any step can significantly affect the results, leading to variations of up to 2% in a sequence of 10 steps. Hence, each transformation in the synthesis flow is important and should be treated carefully for decision-making.

At each step, the model iteratively evaluates candidate transformations within action space $\mathcal{A}$ (referred to as "arms") and

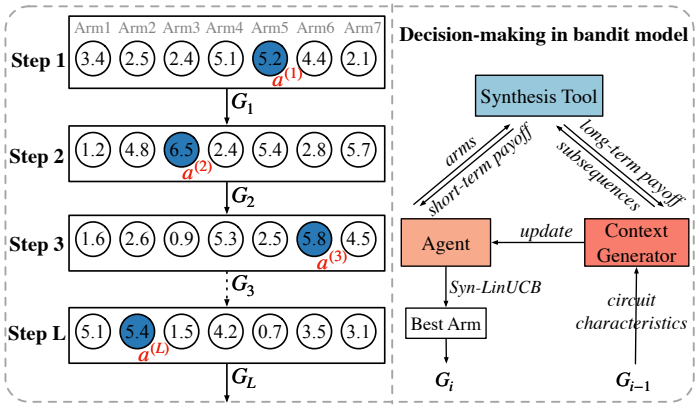
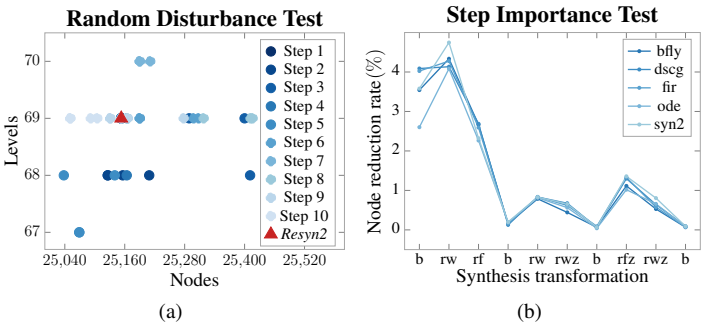


Fig. 3 CBTune framework overview.



(a) (b)

Fig. 4 Analysis of transformation effectiveness in synthesis flow on the VTR 8.0 [11] benchmark.

ultimately selects the highest-scoring one, denoted as $a^{(i)}$, as the decision result. Next, $a^{(i)}$ is applied to the current step's input AIG G_{i-1} to create a new G_i for the subsequent step. The process follows the relation $G_i = a^{(i)} \cdot (G_{i-1})$, where $i = 1, 2, \ldots, L$, $a^{(i)} \in \mathcal{A}$, and G_0 denotes the initial unoptimized AIG. Upon completing all L steps, the final G_L is obtained, representing the synthesis flow as the ordered sequence of transformations $a^{(1)}, a^{(2)}, ..., a^{(L)}$.

The right part of Fig. 3 demonstrates fundamental components and their interactions in the bandit model, which will be detailed in subsequent sections. ABC [1] functions as a synthesis tool. The **context generator** samples potential optimization trends for each arm and collects valuable AIG characteristics as state information, which are used to update the agent's decision parameters. The **agent** employs the Syn-LinUCB algorithm, leveraging both contextual vectors and rewards to score arms, and then selects the one with the highest score as the best arm. Furthermore, various **optimization techniques** are proposed to enhance efficiency and precision in decision-making.

Here are the essential bandit model configurations:

Action Space: To assure a fair comparison, we select transformations commonly employed by prior works [5], [8], [10] within the synthesis tool ABC and represent the action space as discrete actions. Specifically, $\mathcal{A}$ = {*resub* (*rs*), *resub -z* (*rsz*), *rewrite* (*rw*), *rewrite -z* (*rwz*), *refactor* (*rf*), *refactor -z* (*rfz*), *balance* (*b*)}, with a total of $n = 7$ candidate transformations.

Reward: The reward r corresponds to the short-term payoff of each arm. Typically, the number of AIG nodes, logic levels or mapped LUTs serves as the target value, reflecting the transformation's optimization effect. r is obtained by conducting a single-step execution of the respective transformation for each arm, followed by scaling it based on the average target value across all arms. This computation takes place at the beginning of each step's decision-making and remains constant throughout the iterations.

B. Context Generator

Contextual information serves as arm features, offering essential environmental and state insights that assist the agent in making accurate decisions during iterations. This requires constructing a one-dimensional vector $\boldsymbol{x}$ of length d, which includes circuit characteristics $\boldsymbol{x^c}$ and the arm's long-term payoff $\boldsymbol{x^l}$, as detailed in TABLE I. The circuit characteristics act as static contexts computed once per step, while the long-term payoff of each action serves as dynamic contexts continuously updated within each iteration. We normalize the entire vector to ensure smooth convergence during iterations.

C. Agent

Inspired by LinUCB [14], we enhance the traditional MAB method by integrating contextual information, reinforcing its capabilities as the decision-making agent at each step. LinUCB leverages contextual data, including arm and environment features, to guide decision-making, and dynamically modifies the agent's decision parameters. The score for each arm a is calculated using the formula:

$$\begin{aligned} \text{LinUCB}_a &= E(a|\boldsymbol{x}) + \alpha \text{STD}(a|\boldsymbol{x}) \\ &= \boldsymbol{x}^\top \cdot \boldsymbol{\theta}_a + \alpha \sqrt{\boldsymbol{x}^\top \boldsymbol{A}_a^{-1} \boldsymbol{x}}. \end{aligned} \tag{2}$$

Recall that $\boldsymbol{x}$ is the context vector. The first term $E(a|\boldsymbol{x})$ signifies the agent's observed payoff for selecting arm a, determined by the context vector $\boldsymbol{x}$ and decision parameter $\boldsymbol{\theta}_a$. The latter term represents the upper confidence bound, which indicates the standard deviation (STD) between the observed payoff and true payoff, with the derivation process detailed in [15]. Here, α acts as a hyperparameter controlling the exploration level, commonly set to $1 + \sqrt{\ln(2/\delta)/2}$. Each arm maintains such a score, which is continuously updated during the algorithm's iterations.

Considering the mentioned details, we introduce the Syn-LinUCB algorithm, described in Algorithm 1. In each step i, we begin by calculating the reward r_a for each arm and extracting the circuit characteristics $\boldsymbol{x}_a^c$ of G_{i-1}. To address the varying importance of information in the context vector, we adopt a feature importance analysis approach similar to [10] and incorporate a predefined weight vector $\boldsymbol{w}$ to quantify their significance. This weight vector serves as a guide for the agent to assign scores to each arm during iterations. Additionally, we track the number of times each arm is selected with a variable s and use it to calculate the exploration hyperparameter α. Multiple selections of an arm indicate its proximity to the true payoff and result in a reduced exploration level for that arm.

TABLE I Contextual Information.

Feature	Description	Example
Circuit Characteristics ($\boldsymbol{x}^c$)	AIG information extracted by applying $a_j^{(i)}$ ($j \leq n$) to G_{i-1} using circuit characterization tools yosys [12] and ccirc [13].	#Number of wires/cells/nots, #Maximum delay, #Number of combinational nodes, #Number of high degree comb, #Reconvergence, #Node shape, #Fanout distribution, # Edge length distribution.
Long-term Payoff of the Arm ($\boldsymbol{x}^l$)	Random DSE result: During each iteration, "arm" $a^{(i)}$ serves as the first transformation for generating m random subsequences of length l ($i+l \leq L$). These subsequences are then applied to G_{i-1} and get the synthesis results.	Arm: *rewrite* (*rw*); l = 5; m = 3; {**rw**,b,rw,rf,rfz} → Nodes: 28000, Level: 65 {**rw**,rw,rfz,rf,rs} → Nodes: 27890, Level: 66 {**rw**,rf,rf,rw,b} → Nodes: 28010, Level: 66 Arm: *refactor* (*rf*); l = 4; m = 2; {**rf**,b,rf,rw} → Nodes: 28350, Level: 69 {**rf**,rw,b,rs} → Nodes: 28324, Level: 67

Algorithm 1 Syn-LinUCB

Input: Arms $a \in \mathcal{A}$, Context weights $\boldsymbol{w} \in \mathbb{R}^d$, Number of iterations T, Constant ρ.
Output: Best arm a_{best} in this step.

1: $r_a \leftarrow$ Reward of all arms;
2: Extract the AIG characteristics: $\boldsymbol{x}_a^c \in \mathbb{R}^{d_1}$;
3: Arm selection times $s_a = 0$;
4: **for** $t = 1, 2, ..., T$ **do**
5: Update the long-term payoff: $\boldsymbol{x}_{t,a}^l \in \mathbb{R}^{d_2}$;
6: Observe features of $a \in \mathcal{A} : \boldsymbol{x}_{t,a} = [\boldsymbol{x}_a^c, \boldsymbol{x}_{t,a}^l] \in \mathbb{R}^d$;
7: **for** $\forall a \in \mathcal{A}$ **do**
8: Initialize historical context and reward by $\boldsymbol{A}_a = \boldsymbol{I}_d$, $\boldsymbol{b}_a = \boldsymbol{0}_d$, $\forall a$ is new;
9: Update hyperparameter α by $\alpha = 1.0 + \sqrt{\frac{\log(2.0/\rho)}{s_a}}$;
10: Update the decision parameter by $\boldsymbol{\theta}_a = \boldsymbol{A}_a^{-1}\boldsymbol{b}_a$;
11: Calculate the weighted context $\boldsymbol{x}_{t,a}^w = \boldsymbol{x}_{t,a}\boldsymbol{w}$;
12: Update score by $p_{t,a} = \boldsymbol{\theta}_a^\top \boldsymbol{x}_{t,a}^w + \alpha\sqrt{\boldsymbol{x}_{t,a}^{w\top}\boldsymbol{A}_a^{-1}\boldsymbol{x}_{t,a}^w}$;
13: **end for**
14: Choose arm by $a_t = \text{argmax}_{a \in \mathcal{A}}\, p_{t,a}$;
15: Increase the selection count of arm a_t by $s_{a_t} = s_{a_t} + 1$;
16: Update the parameters $\boldsymbol{A}_{a_t}$ and $\boldsymbol{b}_{a_t}$ of the chosen arm a_t by Equation (3);
17: **end for**
18: $a_{best} \leftarrow a_t$.

During each iteration, we first compute the long-term payoffs $\boldsymbol{x}_a^l$ for each arm and construct the corresponding context vector $\boldsymbol{x}_a$ of dimension d by combining the precomputed $\boldsymbol{x}_a^c$. Subsequently, we calculate each arm's score $p_{t,a}$ from line 7 to line 13 using the agent's decision parameters $\boldsymbol{\theta}_a$, the arm's context vector $\boldsymbol{x}_{t,a}^w$ with weight information, and the predefined hyperparameter α. To achieve this, we first derive the decision parameter $\boldsymbol{\theta}_a$ with $\boldsymbol{\theta}_a = \boldsymbol{A}_a^{-1}\boldsymbol{b}_a$, based on the historical context and reward information stored in $\boldsymbol{A}_a$ and $\boldsymbol{b}_a$. If an arm is untested, we initialize its parameters $\boldsymbol{A}$ and $\boldsymbol{b}$ with an identity matrix and a zero vector. Next, we create a context vector $\boldsymbol{x}^w$ with weight information by setting $\boldsymbol{x}_{t,a}^w = \boldsymbol{x}_{t,a} \cdot \boldsymbol{w}$, and calculate the arm's score using Equation (2) in line 12. The arm with the highest score in the current iteration is then selected, and its decision parameters A_{a_t} and $\boldsymbol{b}_{a_t}$ are adjusted based on the context vector and reward of the chosen arm a_t.

$$\boldsymbol{A}_{a_t} = \boldsymbol{A}_{a_t} + \boldsymbol{x}_{t,a_t}\boldsymbol{x}_{a_t}^\top, \quad \boldsymbol{b}_{a_t} = \boldsymbol{b}_{a_t} + r_a \boldsymbol{x}_{t,a_t}. \tag{3}$$

Finally, after T iterations, the arm with the highest score a_{best} is chosen as the final decision.

Syn-LinUCB iteratively updates the agent's scoring parameters based on the arm's contextual information and rewards, offering two key advantages. Firstly, it employs short-term payoffs as the reward, enabling the agent to select arms with the ideal target value at each step, thereby enhancing local performance. Secondly, it mitigates the risk of falling into local optima by considering the arm's long-term payoffs and exploring potential optimization trends, leading to improved decision quality.

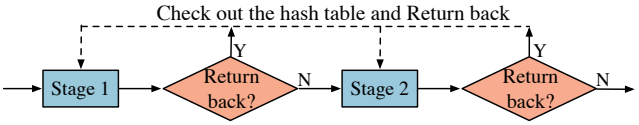


Fig. 5 The return-back mechanism in CBTune.

D. Optimization Techniques

To enhance decision-making performance and reduce the online-learning time, we propose some optimization methods.

Return-back Mechanism: In our framework, each step employs a separate bandit model for decision-making without parameter sharing, potentially leading to suboptimal results due to the lack of historical decision information. To address this, we incorporate a "return-back" mechanism in CBTune. By preserving synthesis results and optimized circuits from previously decided steps, we can assess decision quality after each step and promptly revisit the previous key step for re-decision-making, as depicted in Fig. 5. This adaptive mechanism enhances the final performance by enabling CBTune to learn from mistakes and optimize previous decisions.

To be specific, we create a hash table to store the results of the context generator's sequential multi-step exploration, which takes place during the evaluation of long-term payoffs at each step. The hash table is structured with the first dimension denoting the current step number and the second dimension denoting the various search depths (referred to as l in TABLE I) explored within that step. This allows us to store optimal results for all arms according to the search depths in each step. For example, in step 1, the context generator performs random sampling to explore potential synthesis results in steps 2, 3, and 4. We then identify the optimal value at all these search depths and record these values in the step 1 row of the hash table, aligned with their respective search depths in the columns.

Hereby, after completing each decision-making step, we proceed to compare the current synthesis result with the data

stored in the hash table corresponding to that step. If the difference exceeds a predefined threshold, we record the step number associated with the optimal result for the current search depth in the hash table. Then we revisit the specified step, exclude the previously selected arm from the candidate arm set $\mathcal{A}$, and re-make the decision. Subsequent steps from that step will also be re-executed. Notably, each step can only be revisited once for the sake of efficiency.

Heuristic Tuning Strategy: This strategy explores potential heuristics to boost the agent's decision efficiency in the bandit model. (1) Tuning iteration count. We observe that the initial transformations in the synthesis flow have a substantial influence on the overall results. Fig. 4(b) depicts sequential execution of *resyn2*, tracking the reduction rate of AIG nodes at each step. The initial transformations cut node count by over 3%, whereas subsequent ones yield less than 1%. Therefore, it is inspired to allocate more iterations and runtime to the crucial initial step's decision-making, and this will improve the precision and reliability of the arm scores while preventing the entire decision-making process from veering off course. Simultaneously, as step i progresses, we decrease the iteration count T for subsequent transformations to minimize runtime, following the equation: $T = T_{orig} - decay\_rate \times i$, and $T \geq T_{min}$. (2) Early stop. When the agent repeatedly selects the same arm more than three times in a row during iterations, it indicates the algorithm is converging towards a stable selection result. In such cases, even if the specified number of iterations T has not been reached, we terminate the iteration loop and consider the currently selected arm as the final result.

IV. Experimental Results

The CBTune framework is implemented in C++/C programming language, utilizing Eigen for matrix operations and OpenMP for parallel acceleration. The whole framework is integrated into the source code of ABC, allowing for joint compilation. Execution results are acquired by invoking the "cbtune" command within the interactive interface of ABC. All experiments are conducted on a machine with 40 core Intel® Xeon® Silver 4210R CPU @ 2.40GHz.

We evaluate CBTune by comparing the optimization performance of the synthesis flow it generates with state-of-the-art methods across various benchmarks. These methods can be categorized into two groups: one utilizes lightweight bandit algorithms, and the other employs classical reinforcement learning with complex neural network training. The experiments are conducted within the action space described in Section III-A, aiming to reduce the number of AIG nodes and 6-input LUTs (abbreviated as "6-LUTs").

Comparison with Bandit-based Method: Our approach is compared to FlowTune [8] using the VTR benchmarks [11]. This involves examining the AIG node count following logic minimization and the subsequent assessment of 6-LUTs count after FPGA technology mapping. For FlowTune, we configure "stages:iterations" (s:m) as $3:20$ with 2-repetition, resulting in 14 transformations per stage and a total synthesis flow length of 42. Additionally, we present results from a greedy baseline that selects the best transformation at each decision step. To gain insights into the approximate optimization effect on the overall solution space, we also randomly generate 50,000 synthesis flows for comparison.

TABLE II Details of selected VTR benchmarks [11].

Benchmark	*bfly*	*dscg*	*fir*	*ode*	*or1200*	*syn2*
Nodes	28910	28252	27704	16069	12833	30003
Levels	97	92	94	98	148	93

The experimental results for logic optimization are visualized in Fig. 6, which illustrates the relationship between the AIG node count (x-axis) and the corresponding AIG logic depth (y-axis). For reference, the benchmarks' initial node and level counts can be found in TABLE II. To ensure an equitable comparison, CBTune and the other two methods adhere to FlowTune's implementation by integrating the supplementary subsequence "*ifraig;dch -f*" after every 14 transformations. The results demonstrate CBTune's superior performance over FlowTune, achieving an average improvement of 0.72% in terms of the objective, while also delivering an impressive 39% reduction in runtime. Furthermore, when compared to the Random and Greedy approaches, CBTune exhibits a significant improvement of around 3.6% and 1.6%, respectively, mitigating the risk of encountering local optima.

For reducing 6-LUTs after FPGA technology mapping, results are shown in TABLE III, where #L$\hat{\text{U}}$Ts and #L$\bar{\text{U}}$Ts denotes CBTune's best and average performance. We adapt the supplementary subsequence "*ifraig;scorr;dc2;strash;dch -f;if -K 6;mfs2;lutpack -S 1*" after every 14 transformations, aligning with Flowtune's implementation. CBTune was executed 20 times to gather reliable data. Experimental results show that CBTune surpasses both Greedy and Flowtune in 6-LUTs optimization, yielding average enhancements of 2.9% and 1.4%, respectively, and delivering a notable 54.7% runtime speedup.

Comparison with NN-Enhanced Classical RL Method: We further compare our approach with two other SOTA works: DRiLLS [5] and RL4LS [10], following the same experimental settings targeted reducing the 6-LUTs after FPGA mapping. In contrast to our online-learning bandit method, these works rely on classical reinforcement learning techniques and necessitate pre-trained neural network models to guide their decisions for the synthesis flow. Our experiments use the EPFL benchmark [16] with a fixed synthesis flow length of $L = 25$. We incorporate the priority cuts mapper [17] in ABC, specifically employing the command "*if -a -K 6*" to optimize mapping for 6-LUTs and enhance area efficiency. TABLE IV presents the comparison of results between CBTune and other methods for optimizing 6-LUTs. We extract the optimal performance results reported in [10] for the baseline and calculate their average runtime on our local machine. For CBTune, we execute it ten times to obtain the average target results and the corresponding runtime (τ) for online decision-making. The results demonstrate CBTune's superiority over the other three methods, achieving reductions of 2.2%, 4.4%, and 3.9% in the number of 6-LUTs, respectively. Moreover, CBTune incurs the lowest average tim-

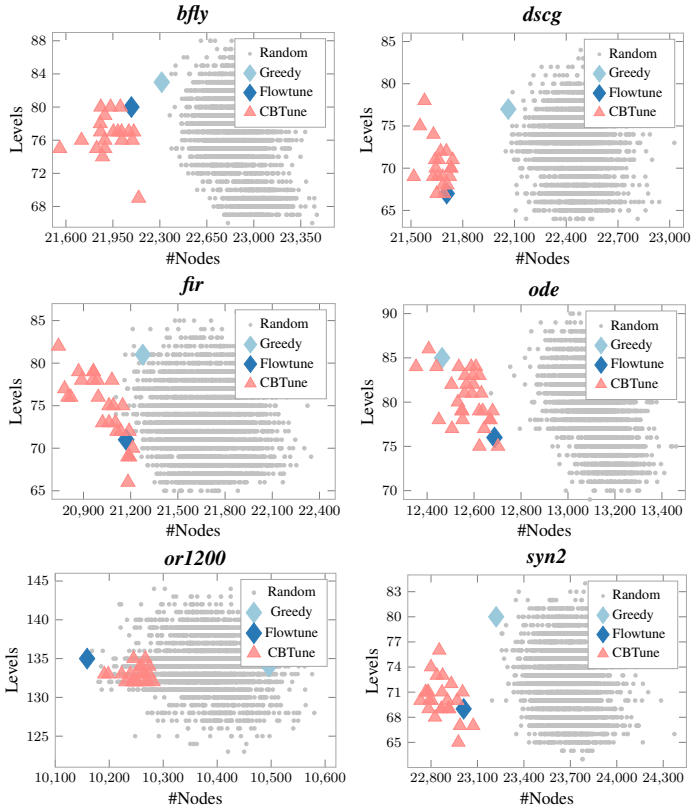


Fig. 6 CBTune vs. FlowTune [8] in AIG node optimization.

TABLE III CBTune vs. FlowTune in 6-LUTs optimization.

Benchmark	Initial	Greedy	Flowtune [8]		CBTune		
	#LUTs	#LUTs	#LUTs	$\tau(m)$	#LUTs	#LUTs	$\tau(m)$
bfly	9019	8269	8216	76.47	**7962**	8086.03	**29.63**
dscg	8534	8313	8302	77.15	**7981**	8119.84	**30.44**
fir	8646	8385	8094	74.23	**7820**	7977.38	**27.6**
ode	5244	5316	5096	34.83	**4920**	5046.71	**17.32**
or1200	2776	2748	2747	20.08	**2731**	2754.07	**15.62**
syn2	8777	8669	8603	81.33	**8234**	8360.53	**31.67**
GEOMEAN	6631.20	6464.69	6364.89	54.04	**6166.39**	6271.82	**24.48**
Ratio Avg.	1.000	0.975	0.960	1.000	**0.930**	0.946	**0.453**

ing cost, approximately 8.37 minutes per design.

In general, our CBTune framework efficiently generates synthesis flows, achieving stable and outstanding optimization results without the need for training sets or complex procedures, while maintaining optimal decision-making runtime.

V. Conclusion

In summary, this work customizes the contextual bandit algorithm to generate efficient synthesis flows, enhancing the circuit design's QoR. The CBTune framework introduces a context generator to support informed decision-making and employs the Syn-LinUCB algorithm as the agent to iteratively evaluate arms, ultimately selecting the optimal one. We also implement optimization techniques like "return-back" mechanism, to prevent falling into local optima and improve decision-making performance. Experimental results highlight CBTune's excellence in optimizing AIG nodes and 6-LUTs within an ideal runtime. Future work will focus on integrating backend information to guide decision-making and further evade staged local optima for QoR.

TABLE IV CBTune vs. NN-enhanced RL in 6-LUTs optimization.

Benchmark	Initial	Greedy	DRiLLS [5]		RL4LS*		CBTune	
	#LUTs	#LUTs	#LUTs	$\tau(m)$	#LUTs	$\tau(m)$	#LUTs	$\tau(m)$
max	721	697	694	32.58	687.8	54.34	**684.25**	**6.01**
adder	249	**244**	**244**	24.05	**244**	10.05	**244**	**5.97**
cavlc	116	115	112.2	26.02	111.3	3.22	**111**	**2.37**
ctrl	29	**28**	**28**	24.25	**28**	2.85	**28**	**0.59**
int2float	47	46	42.6	21.7	42.3	2.81	**40**	**2.76**
router	73	67	70.1	22.01	69.5	3.07	**68.11**	**2.32**
priority	264	146	**133.4**	23.32	142.9	5.9	138.86	**3.41**
i2c	353	291	292.1	25.17	289.32	7.55	**283.11**	**3.61**
sin	1444	1451	1441.5	51.15	**1438**	20.1	1441.67	**9.71**
square	3994	3898	3889.4	130	3889	72.88	**3882.11**	**25.99**
sqrt	8084	4807	4708	147.64	4685.3	196.15	**4607**	**36.51**
log2	7584	7660	7583.6	198.6	7580.1	125.28	**7580**	**41.27**
multiplier	5678	5688	5678	180.84	**5672**	187.81	5679.75	**29.08**
voter	2744	1904	1834.7	84.43	**1678.1**	330.48	1682.25	**11.46**
div	23864	4205	7944.4	259.75	7807.1	482	**4180.91**	**25.58**
mem_ctrl	11631	**10144**	10527.6	229.33	10309.7	1985.84	10242.57	**45.81**
GEOMEAN	926.59	732.69	753.49	59.48	748.34	34.54	**712.83**	**8.37**
Ratio Avg.	1.000	0.791	0.813	1.000	0.808	0.581	**0.769**	**0.141**

* *Last10* in RL-PPO-Pruned [10]

References

[1] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*, 2010.

[2] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. DAC*, 2018.

[3] N. Wu, J. Lee, Y. Xie, and C. Hao, "Lostin: Logic optimization via spatio-temporal information with hybrid graph models," in *Proc. ASAP*, 2022.

[4] C. Yu and W. Zhou, "Decision making in synthesis cross technologies using lstms and transfer learning," in *Proc. MLCAD*, 2020.

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

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

[7] A. B. Chowdhury, B. Tan, R. Karri, and S. Garg, "Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis," *arXiv preprint arXiv:2110.11292*, 2021.

[8] 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 TCAD*, 2022.

[9] W. Jouini, D. Ernst, C. Moy, and J. Palicot, "Upper confidence bound based decision making strategies and dynamic spectrum access," in *Proc. ICC*, 2010.

[10] G. Zhou and J. H. Anderson, "Area-Driven FPGA Logic Synthesis Using Reinforcement Learning," in *Proc. ASPDAC*, 2023.

[11] 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 TRETS*, vol. 7, no. 2, pp. 1–30, 2014.

[12] C. Wolf, "Yosys open synthesis suite," 2016.

[13] M. D. Hutton, J. Rose, J. P. Grossman, and D. G. Corneil, "Characterization and parameterized generation of synthetic combinational benchmark circuits," *IEEE TCAD*, vol. 17, no. 10, pp. 985–996, 1998.

[14] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *The Web Conference*, 2010.

[15] T. J. Walsh, I. Szita, C. Diuk, and M. L. Littman, "Exploring compact reinforcement-learning representations with linear regression," *arXiv preprint arXiv:1205.2606*, 2012.

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

[17] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007.