CB-EVO: Contextual Bandit Tuning with Evolutionary Search for Logic Synthesis

FANGZHOU LIU, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
WUQIAN TANG, Computer Science, National Tsing Hua University, Hsinchu, Taiwan
ZEHUA PEI, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
ZIYANG YU, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
HAISHENG ZHENG, Shanghai Artificial Intelligence Laboratory, Shanghai, China
ZHUOLUN HE, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
MENGJIA DAI, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong
TINGHUAN CHEN, The Chinese University of Hong Kong - Shenzhen, Shenzhen, China
BEI YU, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong

In logic synthesis, pre-optimization involves applying a sequence of transformations, referred to as a synthesis flow, to reduce the complexity of a circuit's Boolean logic graph, such as the And-Inverter Graph (AIG). The primary challenge lies in selecting and ordering these transformations, a task complicated by the exponentially large solution space. In this work, we propose CB-EVO, a novel online learning framework that combines contextual bandit tuning with an evolutionary algorithm to efficiently explore the solution space and generate high-quality synthesis flows. The Syn-LinUCB algorithm functions as the pivotal agent in our bandit tuning process, incorporating circuit-specific characteristics and leveraging long-term payoffs to guide decision-making effectively, thereby circumventing the pitfalls of local optima. To complement the bandit algorithm, we design an evolutionary algorithm that accelerates the decision-making process and enables the efficient expansion of the sequence. Moreover, we explore practical heuristics, such as the "return-back" mechanism and the utilization of a choice network, to further enhance optimization outcomes. Experimental results demonstrate that our framework identifies the optimal synthesis flow with reduced computational time,

Authors' Contact Information: Fangzhou Liu (corresponding author), Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: fzliu23@cse.cuhk.edu.hk; Wuqian Tang, Computer Science, National Tsing Hua University, Hsinchu, Taiwan Province, Taiwan; e-mail: wqtang@cs.nthu.edu.tw; Zehua Pei, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: zhpei23@cse.cuhk.edu.hk; Ziyang Yu, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: yuziyang96@gmail.com; Haisheng Zheng, Shanghai Artificial Intelligence Laboratory, Shanghai, Shanghai, China; e-mail: zhsleo@outlook.com; Zhuolun He, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: zleonhe@gmail.com; Mengjia Dai, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: mjdai@link.cuhk.edu.hk; Tinghuan Chen, The Chinese University of Hong Kong - Shenzhen, Shenzhen, Guangdong, China; e-mail: chentinghuan@cuhk.edu.cn; Bei Yu, Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk.

ACM 1084-4309/2026/06-ART119
https://doi.org/10.1145/3779431

significantly decreasing the number of AIG nodes and LUT count compared to state-of-the-art (SOTA) methods.

CCS Concepts: • **Hardware → Combinational synthesis**; **Circuit optimization;**

Additional Key Words and Phrases: Logic synthesis, bandit algorithm, design space exploration

ACM Reference Format:
Fangzhou Liu, Wuqian Tang, Zehua Pei, Ziyang Yu, Haisheng Zheng, Zhuolun He, Mengjia Dai, Tinghuan Chen, and Bei Yu. 2026. CB-EVO: Contextual Bandit Tuning with Evolutionary Search for Logic Synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 31, 6, Article 119 (June 2026), 26 pages. https://doi.org/10.1145/3779431

1 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 (QoRs).

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 transformations in the sequence hinders efficient logic optimization.

Machine learning is widely used in pre-optimization to accelerate design convergence and minimize manual supervision. LSOracle [3] and HeLO [4] decompose Boolean logic graphs into subcircuits, which are then automatically mapped to the most suitable logic representations (e.g., AIG, MIG, or XAG) through deep neural network-based determination. Following this, a synthesis flow is implemented to perform hetero-DAG logic optimization, which significantly reduces power, performance, and area (PPA) metrics. Emerging methodologies [5–9] suggest using pre-trained models and transfer learning to leverage established synthesis flows for quick PPA predictions, enabling quick selection of the optimal flow from available synthesis flow candidates. This approach involves modeling the circuit structure and the sequential features of transformations within the synthesis flow.

Recently, numerous studies [10–13] have explored synthesis flow generation tasks that utilize online learning to flexibly generate customized sequences for specified design requirements. A representative approach employs reinforcement learning (RL) to guide transformation selection and leverages agent-driven decisions to accelerate solution space exploration. For instance, DRiLLS [10] introduces an intelligent framework enabling an Advantage Actor-Critic (A2C) agent to select optimal transformations flexibly. Zhu et al. [11] utilize Graph Neural Networks (GNNs) to model AIG topologies and integrate historical decision patterns to improve state representations for policy learning. Moreover, to address the challenge of transferring trained RL models across heterogeneous designs, [13, 14] propose a hybrid method combining RL with search-based optimization. Their framework adopts a conditional strategy: pre-trained agents are used for target netlists

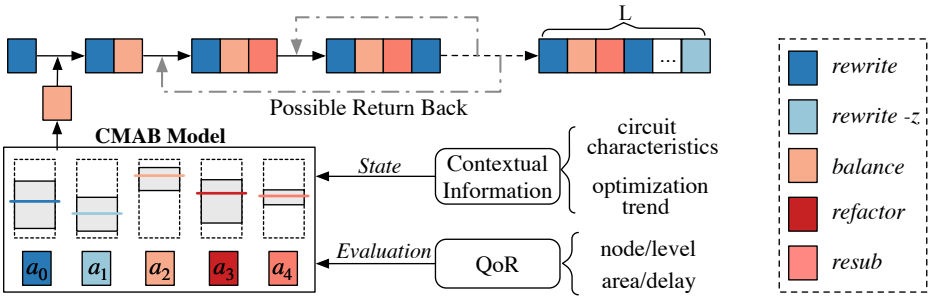


Fig. 1. Illustration of our proposed contextual bandit-based approach for synthesis flow generation. Each color-coded box denotes one arm in the bandit model (i.e., one action in the action space). Uncertainty is represented by the upper confidence bound (UCB; top edge of the grey box). At each step, the arm with the highest UCB is selected and applied to the current AIG, and the process proceeds sequentially.

matching database entries, while Monte Carlo tree search (MCTS) guides exploration for unseen topologies.

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 require substantial time and resources, and achieving model convergence can be challenging. Secondly, model transferability is limited due to reliance on dataset-specific objectives [2, 10], resulting in poor generalizability. OpenABC-D [15] reveals that the similarity between optimal synthesis flows across different circuits is less than 30%, underscoring the design-specific nature of synthesis flow generation and impeding model transferability. In response, BOiLS [16] suggests using Bayesian optimization to combine sequence similarity modeling with dynamic local search. This approach effectively addresses the curse of dimensionality in high-dimensional search spaces for synthesis flows and allows for on-the-fly, design-specific searches without the need for pre-training. Expanding upon this, the lightweight multi-armed bandit (MAB) model in FlowTune [17, 18] further simplifies the decision-making process in online learning and enhances computational efficiency. The bandit algorithm, based on a trial-and-error mechanism, balances exploration and exploitation to maximize overall payoffs. Unlike Bayesian optimization, this approach adapts strategies based on real-time feedback and does not require complex calculations of posterior distributions. Hence, it can avoid the intricacies of constructing probabilistic models, making it more effective in scenarios that demand frequent decision updates.

Drawing inspiration from previous works, in this article, we introduce "**CB-EVO**", a novel online learning framework that leverages a contextual bandit approach enhanced with evolutionary algorithms (EAs) for efficient synthesis flow generation. Considering FlowTune as a non-contextual MAB method, its strategy updates rely solely on the real-time synthesis results from sampling tests across arms, neglecting critical arm features specific to the domain, such as optimization trends and current AIG's characteristics. Furthermore, this method makes decisions on a segment-by-segment basis, overlooking permutations within each segment sequence, which comes at the cost of final performance. Therefore, in the critical initial steps of the synthesis flow, we implement a customized contextual bandit algorithm, as illustrated in Figure 1. This approach builds contextual multi-armed bandit (CMAB) models for each transformation decision step within the sequence. Each transformation candidate in the action space maps to a bandit arm. At each step, the bandit uses the current state and immediate evaluation to iteratively select the best arm, applies the corresponding transformation to the current AIG, and then proceeds to the next decision step. Additionally, since our model limits efficiency by selecting only one transformation at each decision step within the

sequence, we incorporate EAs to expedite the extension process after a certain sequence length. Moreover, we empirically implement heuristic techniques, including the "return-back" mechanism and lossless synthesis, to further improve the decision-making results. Our main contributions are summarized as follows:

- Our proposed framework, CB-EVO, addresses synthesis flow generation by adapting contextual bandit algorithms for intelligent transformation selection through iterative model tuning, combined with evolutionary strategies for accelerated sequence extension.
- In the CMAB model, we implement the Syn-LinUCB algorithm as the agent and establish a context generator for informed decision-making within the bandit model.
- We introduce several optimization techniques: a novel "return-back" mechanism that revisits decisions to escape local optima, distinguishing it from traditional RL scenarios, and the incorporation of the CHOICE command into the sequence to retain optimization potential for subsequent transformations.
- Experimental results show that our framework outperforms SOTA approaches within the same action space.

The remainder of this article is organized as follows. Section 2 provides the necessary background on synthesis flow generation and bandit algorithms, explaining their relevance to the problem at hand. Section 3 presents the CB-EVO framework, elaborating on how the contextual bandit method balances exploration and exploitation, and how the EA improves decision-making efficiency. Section 4 outlines the optimization techniques employed in our algorithm to enhance PPA outcomes. Section 5 analyzes the experimental results and their implications. Finally, Section 6 concludes the article and proposes potential directions for future research.

2 Preliminaries

2.1 Boolean Logic Optimization

Boolean logic optimization employs techniques like logic sharing and reusing to minimize Boolean networks. The and-inverter graph 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 [19, 20]. 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.

2.2 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 the overall payoff. One prominent MAB algorithm, Upper Confidence Bound (UCB) [21], ingeniously devises a tradeoff 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,

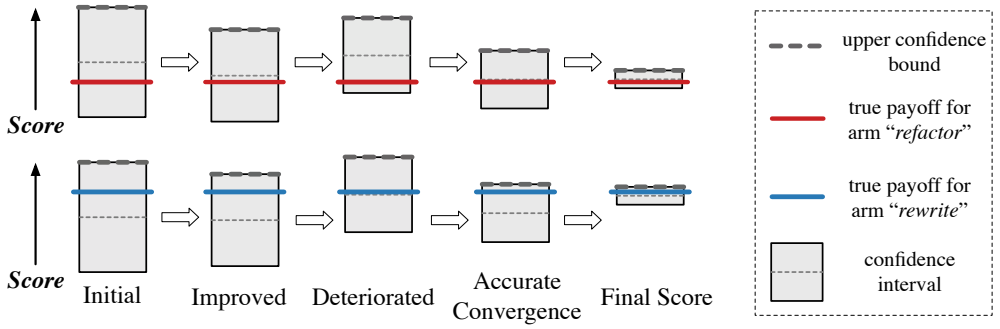


Fig. 2. Score iterations of the *rewrite* and *refactor* arms in the bandit model. Red and blue lines denote each arm's (unknown) true payoff. The grey box indicates each arm's confidence interval. The thick grey dashed line is the upper confidence bound; the thin grey dashed line is the empirical mean.

leading to $p' \to p$. The UCB_i (i.e., p') for the ith arm in iteration t is given by

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

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 a higher upper bound and fewer trials to those with a lower upper bound.

2.3 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 Figure 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 by the application of each arm (i.e., a single transformation a), without foreseeing its long-term payoff in the total flow, which will be elaborated upon in Section 3.1. 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. Transient deviations may occur—the UCB can temporarily depart from the true payoff—but as iterations proceed, confidence intervals shrink and the UCB concentrates near the true payoff, reducing suboptimal selections. After sufficient iterations for convergence, informed choices can be made based on each arm's score.

3 CB-EVO Framework

This section presents the proposed CB-EVO framework and its algorithmic details. As illustrated in Figure 3, our modeling framework integrates a CMAB model based on the contextual bandit

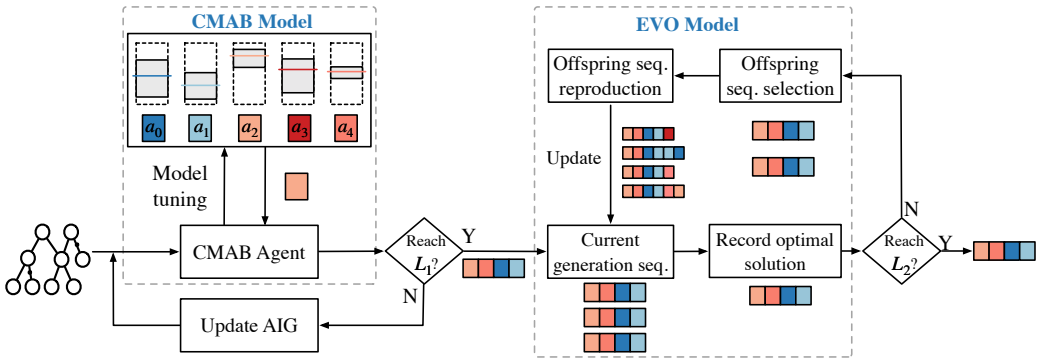
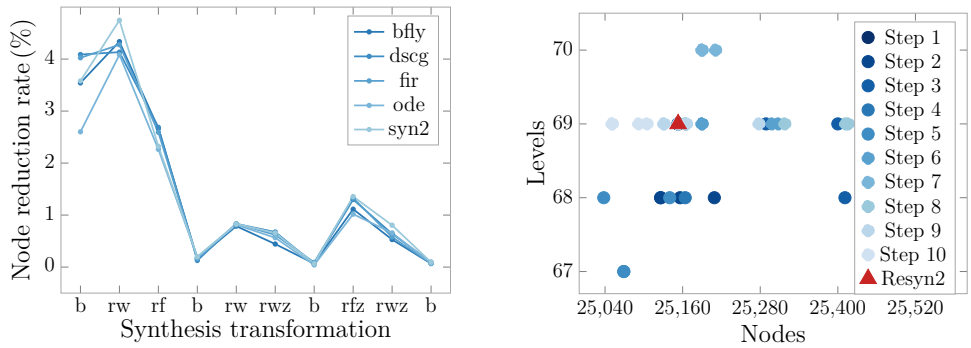


Fig. 3. CB-EVO framework overview.



(a) Step importance test (b) Random disturbance test

Fig. 4. Analysis of transformation effectiveness in synthesis flow using VTR 8.0 [22] benchmark suite.

algorithm with an enhanced EVO model utilizing the EA. This design originates from key observations in synthesis flow analysis: Initial transformations typically exert more significant optimization effects on AIGs, and simultaneously, imprecise selection of these initial transformations may lead to premature convergence to local optima. To support these observations, Figure 4(a) shows the predefined synthesis flow `resyn2`'s application across designs, tracking AIG node reduction rates at each step. The results confirm that initial transformations yield substantially higher reduction rates. Figure 4(b) further investigates `resyn2` through a perturbation test, selectively replacing individual transformations (from action space) while keeping others fixed, and calling the ABC to validate the outcomes. Results show that any single perturbation significantly affects outcomes (up to 2% variation in 10-step sequences), with early-step changes causing the largest deviations.

In the synthesis flow generation task, we define a segment length L_1 and construct independent CMAB models for each transformation decision from step 1 to L_1 using contextual bandit tuning. Crucially, each step's CMAB model makes self-contained decisions without inter-step parameter sharing or communication. Unlike FlowTune [18]'s segment-by-segment bandit algorithm approach, our step-by-step strategy enables each CMAB model to iteratively select optimal transformations from the action space. While this increases runtime, it prioritizes finding high-quality initial sequences and prevents premature convergence to local optima. Upon reaching L_1, we switch to an EVO model that uses the CMAB-generated sequence as an initial population. The model then

Table 1. Key Framework Parameters

Notation	Description	Parameter Scope
L_2	Total target length of the synthesis flow	Global
$\mathcal{A}$	Action space (set of 7 available logic synthesis transformations)	
G	AIG, where G_0 denotes the initial graph	
$\mathbb{F}$	Synthesis flow, where $\mathbb{F}_{L_2}$ denotes the final outcome	
L_1	CMAB model's decision horizon (steps 1 to L_1)	CMAB
$a^{(i)}$	Selected transformation action at step i with $a \in \mathcal{A}$	
r_a	Reward function for arm evaluation	
T	Number of CMAB iterations	
x	d-dim context vector (circuit characteristics x^c and long-term payoff x^l)	
M	Maximum transformations appended per EVO offspring generation	EVO
j	Actual transformations applied per offspring, where $j \in [1, M]$	
t	Top-t offspring sequences retained per generation	

extends the sequence by randomly appending 1 to M transformations to create offspring sequences, which undergo iterative testing and selection until reaching the target length L_2.

To assure a fair comparison, we adopt transformations commonly used in prior works [10, 18, 23] within the logic synthesis tool ABC, representing the action space as discrete operations. Specifically, we define the action space $\mathcal{A}$ for our CB-EVO framework as: $\mathcal{A}$ = {`resub` (`rs`), `resub -z` (`rsz`), `rewrite` (`rw`), `rewrite -z` (`rwz`), `refactor` (`rf`), `refactor -z` (`rfz`), `balance` (`b`)}, comprising $n = 7$ candidate transformations. For clarity, the key parameters used in this section are listed in Table 1.

3.1 Contextual Bandit Tuning

We first present the CMAB model, which employs a contextualduring the synthesis flow. To construct a synthesis flow of length L_1, we instantiate a new CMAB model at each step to select a transformation and update the current AIG. This process can be formally described as follows:

Definition 1 (Sequence Decision-making Flow). At each step i ($1 \le i \le L_1$), a step-local CMAB model selects a transformation $a^{(i)} \in \mathcal{A}$ to apply to G_{i-1}, producing G_i. The procedure starts from the initial circuit G_0 and terminates at G_{L_1} after L_1 decision steps. The complete synthesis flow is defined as an ordered sequence of transformations: $\mathbb{F}_{L_1} = \langle a^{(1)}, a^{(2)}, \dots, a^{(L_1)} \rangle$, where the final G_{L_1} is obtained through iterative application of the selected transformations:

$$G_{L_1} = \left(a^{(L_1)} \circ \cdots \circ a^{(1)} \right)(G_0). \tag{2}$$

At each step, contextual bandit tuning follows an iterative decision process (see Figure 5): Syn-LinUCB serves as the decision agent, runs for an iteration budget of T rounds, and continually updates its parameters. Note that each transformation $a \in \mathcal{A}$ corresponds to a distinct arm in the bandit model. The operational pipeline comprises three key modules:

(1) **Context Generator**: Extracts real-time AIG characteristics and evaluates potential optimization trends for each bandit arm. These characteristics provide essential state information for updating the agent's decision parameters.
(2) **Agent**: Implements the Syn-LinUCB algorithm to evaluate transformation candidates using both contextual features and historical rewards. It selects the optimal transformation based on a combined score balancing exploration and exploitation.
(3) **Synthesis Tool**: Utilizes ABC [1] to execute selected transformations, compute reward metrics, and supply synthesis results that assist the agent in decision-making.

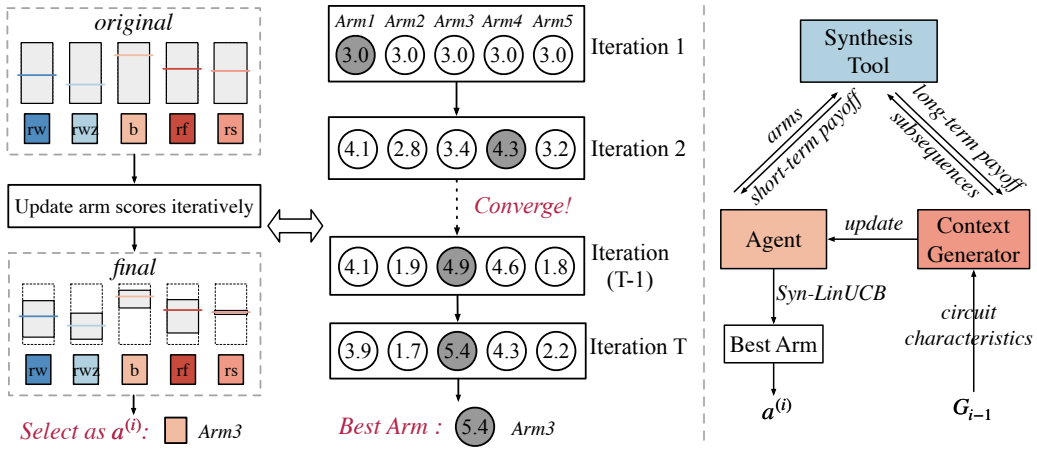


Fig. 5. Details of contextual bandit tuning in one decision step. Each transformation $a \in \mathcal{A}$ corresponds to a distinct arm in the bandit model. The left part of the figure depicts the iterative UCB update process, while the right part demonstrates the interactions among the three key modules.

Table 2. Contextual Information x_a for Each arm $a \in \mathcal{A}$.The Numbers in Parentheses Indicate the Weights $w$used for Each Feature

Feature	Description	Example
Circuit Characteristics (x_a^c)	AIG information extracted by applying "arm" a to G_{i-1} using circuit characterization tools yosys [24] and ccirc [25].	#Number of wires/cells/nots (0.083), Maximum delay (0.014), #Number of combinational nodes (0.089), #Number of high degree comb (0.001), Reconvergence (0.121), Node shape (0.021), Fanout distribution (0.100), Edge length distribution (0.008).
Long-term Payoff of the Arm (x_a^l)	Random DSE result: During each iteration, we use "arm" a as the first transformation to generate m random subsequences of length l ($i + l \leq L_1$), apply them to G_{i-1}, and obtain synthesis results. For each l, we calculate the normalized mean and minimum of these m results, and combine them using the weights $(w^{(l)}_{\text{mean}}, w^{(l)}_{\text{min}})$. The weighted values from all R lengths are concatenated to form the $2R$-dimensional feature vector x_a^l.	For arm `rewrite (rw)`: · When $l = 5$, $m = 3$, $w^{(l)} = (0.75, 0.25)$. {**rw**, b, rw, rf, rfz} → Nodes: 25467, Level: 80; {**rw**, rw, rfz, rf, rs} → Nodes: 25570, Level: 95; {**rw**, rf, rf, rw, b} → Nodes: 25882, Level: 84. · When $l = 4$, $m = 2$, $w^{(l)} = (0.55, 0.45)$. {**rw**, b, rf, rw} → Nodes: 25491, Level: 80; {**rw**, rw, b, rs} → Nodes: 26207, Level: 82r.

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 x of length d, which includes circuit characteristics x^c and the arm's long-term payoff x^l, as detailed in Table 2. 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. To better capture the varying significance of each feature, we adopt a feature importance analysis approach inspired by [23], and introduce a predefined weight vector w to quantify feature importance. For circuit characteristics x_a^c, we use a random forest classifier to identify and retain only the most relevant features, pruning the feature set accordingly. For the long-term payoff x_a^l, we empirically select different parameter values $(w^{(l)}_{\text{mean}}, w^{(l)}_{\text{min}})$ for each l, as exemplified in Table 2.

Agent. Inspired by LinUCB [26], we enhance the traditional MAB method by integrating contextual information, reinforcing its capabilities as the decision-making agent. 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:

$$\begin{aligned} \text{LinUCB}_a &= E(r|x_a) + \alpha \text{STD}(r|x_a) \\ &= x^\top \cdot \boldsymbol{\theta}_a + \alpha\sqrt{x^\top A_a^{-1} x}. \end{aligned} \tag{3}$$

Recall that x is the context vector. The first term $E(r|x_a)$ signifies the agent's observed payoff for selecting arm a, determined by the context vector 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 [27]. Here, α acts as a hyperparameter controlling the exploration level, commonly set to $1 + \sqrt{\ln(2/\rho)/s_a}$. This choice upper-bounds the theoretical confidence radius up to a constant factor, and it decreases monotonically as the number of times arm a has been selected, s_a, increases. The parameter ρ denotes the confidence level, representing the maximum allowed probability that the true payoff is not contained within the computed confidence interval. A smaller ρ leads to a wider confidence bound (more exploration), whereas a larger ρ yields a tighter bound (more exploitation). We set $\rho \in [0.05, 0.15]$ by default. Each arm maintains such a score, which is continuously updated during the algorithm's iterations.

We now justify that the LinUCB score in Equation (3) upper-bounds the true expected payoff with high probability. Assume bounded rewards $r \in [0, 1]$ and a linear stochastic model $\mathbb{E}[r \mid x_a] = x^\top \boldsymbol{\theta}_*$ with sub-Gaussian noise. Let $\hat{\boldsymbol{\theta}}_a = A_a^{-1} b_a$ denote the ridge estimator, where $A_a = \lambda I_d + \sum_{\tau \le t,\, a_\tau = a} x_{\tau,a} x_{\tau,a}^\top$ and $b_a = \sum_{\tau \le t,\, a_\tau = a} r_{\tau,a} x_{\tau,a}$ (we set $\lambda = 1$ in practice). Here, $x_{\tau,a} \in \mathbb{R}^d$ denotes the context vector of arm a at round τ, composed of its static circuit features and dynamically updated long-term payoffs. By self-normalized concentration inequalities for linear bandits (e.g., [26, 27]), there exists a radius $\beta_t(\delta)$ such that, for any iteration t and any arm a,

$$\Pr\left\{ \left|x^\top(\hat{\boldsymbol{\theta}}_a - \boldsymbol{\theta}_*)\right| \;\le\; \beta_t(\delta)\sqrt{x^\top A_a^{-1} x} \right\} \;\ge\; 1 - \delta. \tag{4}$$

Consequently, choosing $\alpha \ge \beta_t(\delta)$ yields the one-sided bound:

$$\Pr\left\{ \underbrace{x^\top \boldsymbol{\theta}_*}_{\text{true payoff}} \;\le\; \underbrace{x^\top \hat{\boldsymbol{\theta}}_a + \alpha\sqrt{x^\top A_a^{-1} x}}_{\text{LinUCB score}} \right\} \;\ge\; 1 - \delta. \tag{5}$$

Moreover, to obtain an *anytime* guarantee over all rounds and arms, set $\delta_t = \delta/(nT)$ and apply a union bound across $t \le T$ and n arms; then, with probability at least $1 - \delta$, the true payoff of every arm at every round is upper-bounded by the LinUCB score in Equation (3) when using $\alpha_t \ge \beta_t(\delta/(nT))$. This parallels the classical Hoeffding-style UCB argument: the first term is an empirical estimate, and the second is a shrinking confidence radius because A_a grows (hence A_a^{-1} contracts) as evidence accumulates. In practice, as mentioned above, we use a decreasing schedule for α, setting $\alpha = 1 + \sqrt{\log(2/\rho)/s_a}$.

Considering the mentioned details, we introduce the Syn-LinUCB algorithm, described in Algorithm 1. To begin with, we compute the reward r_a for each arm, the reward r_a quantifies the effectiveness of a transformation through a metric termed the short-term payoff of each arm a:

$$r_a = \frac{\Delta\mathcal{M}(G_{i-1}, a \circ G_{i-1})}{\mathbb{E}[\Delta\mathcal{M}]}, \quad \mathcal{M} \in \textit{objectives}, \tag{6}$$

ALGORITHM 1: Syn-LinUCB

Input: Arms $a \in \mathcal{A}$, Context weights $w \in \mathbb{R}^d$,
Number of iterations T, Constant ρ.
Output: Selected arm a_{best} for step i (i.e., $a_{\text{best}} = a^{(i)}$).
1: $r_a \leftarrow$ Reward of all arms;
2: Extract the AIG characteristics: $x_a^c \in \mathbb{R}^{d_1}$;
3: Arm selection times $s_a = 0$;
4: **for** $t = 1, 2, ..., T$ **do**
5: Update the long-term payoff: $x_{t,a}^l \in \mathbb{R}^{d_2}$;
6: Observe features of $a \in \mathcal{A}$: $x_{t,a} = [x_a^c, x_{t,a}^l] \in \mathbb{R}^d$;
7: **for** $\forall a \in \mathcal{A}$ **do**
8: Initialize historical context and reward by $A_a = I_d$, $b_a = 0_d$, $\forall a$ is new;
9: Update hyperparameter α by $\alpha = 1.0 + \sqrt{\frac{\log(2.0/\rho)}{s_a}}$;
10: Update the decision parameter by $\boldsymbol{\theta}_a = A_a^{-1} b_a$;
11: Calculate the weighted context $x_{t,a}^w = x_{t,a} w$;
12: Update score by $p_{t,a} = \boldsymbol{\theta}_a^\top x_{t,a}^w + \alpha\sqrt{x_{t,a}^{w\top} A_a^{-1} x_{t,a}^w}$;
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 A_{a_t} and b_{a_t} of the chosen arm a_t by Equation (7);
17: **end for**
18: $a_{\text{best}} \leftarrow a_T$.

where $\Delta\mathcal{M}$ denotes the reduction in a target metric (e.g., AIG nodes, logic levels, or LUT count), reflecting the optimization impact of the transformation, and $\mathbb{E}[\Delta\mathcal{M}]$ represents the average improvement across all arms. The reward r_a is computed by performing a single-step execution of the corresponding transformation for each arm and normalizing its outcome against the mean improvement of all arms. This calculation occurs once at the start of each decision step and remains fixed throughout subsequent iterations within the same decision step.

After reward computation, we proceed to extract the circuit characteristics x_a^c from G_{i-1} for each arm. To reflect the varying significance of features, we assign predefined weights w based on feature importance analysis (see Section 3.1), which guide the agent in scoring each arm. 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.

With both the reward and static features in place, we enter the iterative decision process. At each iteration, we first compute the long-term payoffs x_a^l for each arm and construct the corresponding context vector x_a of dimension d by combining the precomputed x_a^c. We then calculate the score $p_{t,a}$ for each arm (as shown in lines 7–13 of the algorithm) using the agent's decision parameters $\boldsymbol{\theta}_a$, the weighted context vector $x_{t,a}^w$, and the exploration parameter α. Specifically, the decision parameter $\boldsymbol{\theta}_a$ is updated as $\boldsymbol{\theta}_a = A_a^{-1} b_a$, based on the historical context and reward information stored in A_a and b_a. This equation can be regarded as the ridge estimate of $E(r \mid x_a)$, which enables efficient online updates and a principled UCB construction. If an arm is untested, A_a and b_a are initialized to the identity matrix and zero vector, respectively. Next, we create a context vector x^w with weight information by setting $x_{t,a}^w = x_{t,a} \cdot w$, and calculate the arm's score using Equation (3) in line 12. The arm achieving the highest score in the current iteration is then selected. Subsequently, its decision parameters, A_{a_t} and b_{a_t}, are updated based on the context vector and reward of the chosen arm a_t:

$$A_{a_t} = A_{a_t} + x_{t,a_t} x_{t,a_t}^\top, \quad b_{a_t} = b_{a_t} + r_{a_t} x_{t,a_t}. \tag{7}$$

This update follows the LinUCB/regularized least-squares rationale. For each arm a, A_a accumulates the (regularized) design covariance via outer products $x_{t,a}\, x_{t,a}^{\top}$, and b_a accumulates the reward-context cross term via $r_{a_t} x_{t,a_t}$. Finally, after T iterations, the arm a_T with the highest score in the last iteration is selected as a_{best}, representing 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 i, 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.

3.2 EVO Model: Evolutionary Optimization

The step-by-step decision-making approach effectively mitigates the risk of quickly converging to local optima during the critical selection of initial transformations in the sequence. However, this method incurs significant computational and time overhead. To address this, we propose the EVO model, which leverages the output sequence from the CMAB model as the starting point for the EA. By incorporating principles of selection, reproduction, and recombination, the EVO model rapidly extends the sequence. Notably, in this process, each generation no longer produces a single transformation but instead generates a segment sequence composed of multiple transformations. While this approach introduces certain limitations to the optimization results, it significantly enhances decision-making efficiency.

EAs utilize population-based search mechanisms—selection, crossover, and mutation—to escape local optima in complex design spaces. Unlike conventional EAs that initialize populations randomly, our EVO model seeds the evolutionary process with high-quality transformation sequences from the CMAB model, addressing computational bottlenecks in stepwise approaches. EAs iteratively evolve solutions through fitness evaluation and genetic operations, proving particularly effective for complex, non-linear problems where gradient-based methods struggle [28]. This approach is well-suited for synthesis flow generation, where strong epistatic relationships exist between transformations. By focusing on sequence extension rather than point optimization, the EVO model achieves linear time complexity relative to target length L_2.

As depicted in Figure 6, the EVO model initiates its workflow by replicating the contextual bandit-tuned output sequence to form the initial generation. Given the computational inefficiency of repeatedly applying sequences to G_0, we instead replicate t instances of G_{L_1} and perform backward sequence extension from these graphs. For each replicated G_{L_1}, we stochastically apply j synthesis transformations ($j \in [1, M]$), generating updated graphs G_{L_1+1}. From the perspective of sequence generation, this process creates randomized segment sequences composed of multiple synthesis transformations, which serve as mutational variations within the EA. Each iteration produces offspring sequences by extending these segments, forming the initial population for subsequent iterations. Through competitive selection based on synthesis results evaluated against specified objectives, the top-t highest-scoring candidates are retained for further extension via appended random segment sequences. This iterative cycle of evaluation, selection, and mutation-driven reproduction progressively elongates the sequence while optimizing performance metrics. The evolutionary loop terminates when the total sequence attains the predefined target length L_2, at which point the framework outputs both the optimized synthesis flow F_{L_2} and its corresponding AIG G_{L_2} as the final result of the CB-EVO pipeline.

4 Optimization Techniques

This section introduces three synergistic strategies aimed at enhancing the efficacy of online learning while preserving the quality of decision-making. For the CMAB model, we propose a *return-back*

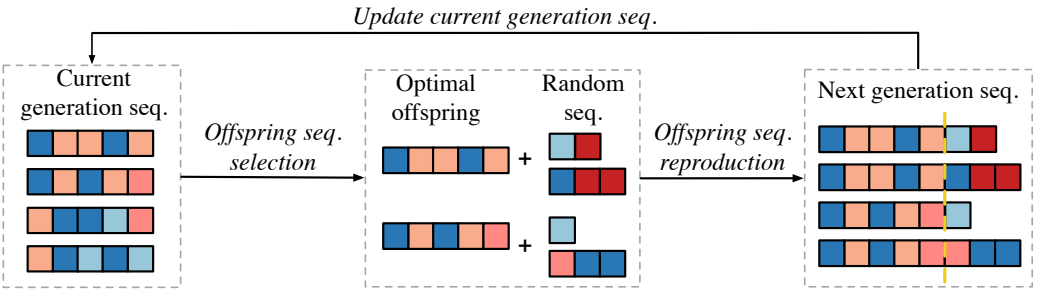


Fig. 6. Workflow of our EVO model.

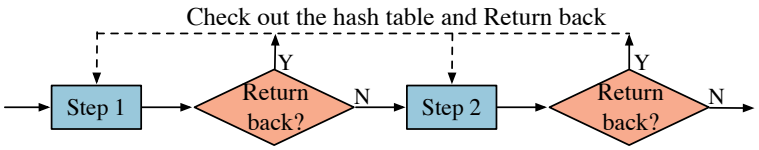


Fig. 7. The return-back mechanism in the CMAB model.

mechanism to facilitate error recovery, coupled with an *adaptive tuning* method that enables systematic coordination of parameter adjustments and computational budgeting. Additionally, for the EVO model, we employ a *lossless synthesis* approach designed to maintain the optimization potential of AIG during the generation of flows.

4.1 Return-back Mechanism

In our CMAB model, 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 introduce a "return-back" mechanism, which diverges from conventional scenarios in RL, and can help us escape the local optimal trap. 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 Figure 7. This adaptive mechanism enhances the final performance by enabling the CMAB model 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 2) 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 2–4. 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 compare the current synthesis result with the data stored in the hash table corresponding to that step, and compute the relative degradation $\Delta = \frac{c_{\text{curr}} - c_{\text{best}}}{c_{\text{best}}} \times 100\%$ (where c_{curr} is the current cost and c_{best} is the best cost recorded for that step). If Δ exceeds a predefined threshold, we record the step number associated with the optimal result for the current search depth in the hash table. Note that this return-back threshold is set

empirically and is adapted to the circuit scale: for larger circuits (with more AIG nodes and higher per-transformation runtime), a higher threshold is used to limit the frequency of revisits, while for smaller circuits, a lower threshold encourages more timely corrections. 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.

4.2 Adaptive Tuning in CMAB Model

This strategy explores potential heuristics to boost the agent's decision efficiency in the CMAB model.

Iteration Budget Scheduling. We observe that the initial transformations in the synthesis flow have a substantial influence on the overall results. Figure 4(a) depicts the 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: $T = \lfloor T_{orig} - i \times decay\_rate \rfloor$, and $T \geq T_{min}$. The parameter $decay\_rate$ controls the rate at which the iteration budget decreases as the algorithm progresses. Typically, $decay\_rate$ is set to a small positive value less than 1, so T drops slowly at each step. In practice, $decay\_rate$ can be chosen based on task complexity and arm count, with the goal of reducing T to T_{min} within the expected number of steps. The use of the ceiling function ensures T remains an integer. For the minimum allowed bandit iterations T_{min}, it is advisable to set it slightly larger than the number of arms to ensure each arm has a reasonable chance to be explored, thus improving the robustness of the selection process. Empirically, in our experiments, we typically set $T_{orig} = 15$, $T_{min} = 8$, and $decay\_rate = 0.4$.

Convergence-driven 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.

4.3 Lossless Synthesis

Lossless synthesis [29] is a technique that preserves intermediate networks generated during technology-independent synthesis by combining them into a single subject graph with choices, enabling the mapper to select optimal subcircuits from multiple structurally distinct but functionally equivalent representations. This is achieved by leveraging combinational equivalence-checking methods [30] (e.g., simulation and SAT) to identify functionally equivalent internal nodes across networks, which are then merged into a unified choice network. During technology mapping, cuts are computed over equivalence classes of nodes, allowing the mapper to dynamically choose the best decomposition from the combined network. The integration of choice commands in synthesis flow generation ensures that critical optimization opportunities in AIGs are preserved, avoiding over-commitment to heuristic transformations that might discard advantageous structures. This approach mitigates structural bias, ensures no degradation in synthesis outcomes, and provides flexibility by combining networks optimized for different objectives (e.g., delay vs. area) into a single framework.

We implement the algorithm proposed in [29] within ABC's command framework as two novel commands: `&choice_store` and `&choice_compute`. The `&choice_store` command temporarily

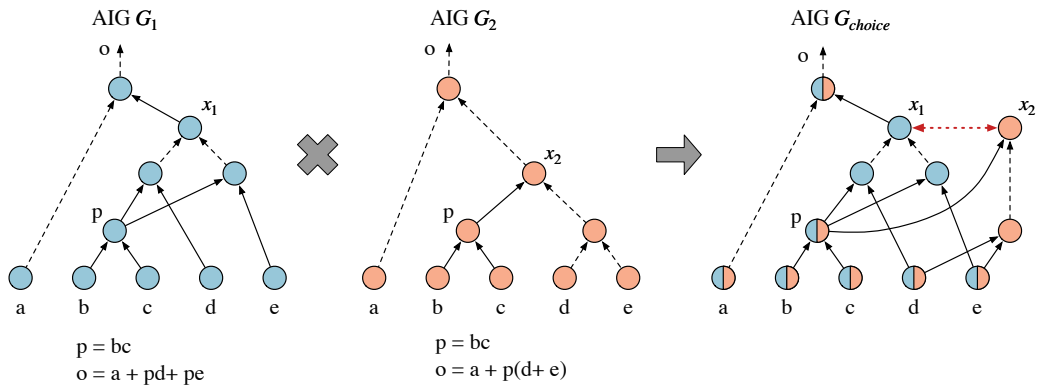


Fig. 8. Creation of an AIG with choices [29]. A choice is created when there are two nodes with the same global function (up to complementation) but different structures. Thus, nodes x_1 and x_2 lead to a choice, but node p does not (p is structurally the same in both networks).

captures and stores the current state of the AIG. After accumulating **three** such AIG snapshots, the `&choice_compute` command merges them into a unified graph and computes choices. It is important to note that in ABC, `&choice_compute` is invoked if and only if exactly three snapshots have been stored; any other count will not trigger the command. This merging process maintains strict logical equivalence while preserving the unique structural signatures of each intermediate AIG stage—thereby ensuring maximal retention of optimization potential across synthesis phases. A simple example is illustrated in Figure 8, where the workflow follows:

$$G_1 \xrightarrow{\texttt{\&choice\_store}} \texttt{rs} \rightarrow G_2 \xrightarrow{\texttt{\&choice\_store}} \texttt{\&choice\_compute} \rightarrow G_{\text{choice}} \rightarrow \begin{array}{c}\text{Further}\\ \text{transformations}\end{array}.$$

This combined representation enables downstream transformations to leverage the full spectrum of AIG characteristics for further refinement. These CHOICE commands are seamlessly integrated into the synthesis flow as additional steps, strategically inserted and executed at appropriate points in the sequence to enhance optimization flexibility without disrupting the original workflow.

5 Experimental Results

We evaluate the performance and properties of the proposed method to answer the following research questions:

(1) How do CB-EVO compare in performance with the state-of-the-art (SOTA) bandit method (Section 5.2)?
(2) How does the performance of CB-EVO measure against classical RL-motivated methods (Section 5.3)?
(3) How does lossless synthesis contribute to escaping local optima and expanding the solution space during LUT optimization (Section 5.4)?
(4) What fundamental tradeoffs exist between LUT minimization and delay reduction when applying distinct optimization objectives in CB-EVO (Section 5.6)?

5.1 Experimental Setup

Environment. The CB-EVO framework is developed in C++/C programming language, utilizing Eigen for matrix operations and OpenMP for parallel acceleration. The implementation is primarily

Table 3. Details of Selected VTR Benchmarks [22]

Benchmark	Inputs	Outputs	AIG Nodes	AIG Levels	#LUTs	Lvls
bfly	482	257	28,910	97	9,023	21
dscg	418	257	28,252	92	8,539	21
fir	450	225	27,704	94	8,659	23
ode	275	169	16,069	98	5,246	23
or1200	588	509	12,833	148	2,776	27
syn2	450	321	30,003	93	8,773	22

Table 4. Details of Selected EPFL Benchmarks [31]

Benchmark	Inputs	Outputs	AIG Nodes	AIG Levels	#LUTs	Lvls
max	512	130	2,865	287	720	117
adder	256	129	1,020	255	249	121
cavlc	10	11	693	16	116	6
ctrl	7	26	174	10	29	2
int2float	11	7	260	16	47	5
router	60	30	257	54	73	17
priority	128	8	978	250	266	105
i2c	147	142	1,342	20	353	7
sin	24	25	5,416	225	1,450	75
square	64	128	18,484	250	3,994	122
sqrt	128	64	24,618	5,058	8,085	3,955
log2	32	32	32,060	444	7,588	160
multiplier	128	128	27,062	274	5,680	126
voter	1,001	1	13,758	70	2,770	37
div	128	128	57,247	4,372	23,865	2,117
mem_ctrl	1,204	1,231	46,836	114	11,638	53

conducted independently of the ABC source code, with the exceptions of the `&choice_store` and `&choice_compute` commands, which are seamlessly integrated into ABC's existing codebase. We use CMake for CB-EVO's code organization and executable compilation, and directly invoke ABC's executable for optimization testing. All experiments are performed on a machine with 40 core Intel® Xeon® Silver 4210R CPU @ 2.40GHz.

Datasets. To ensure a fair evaluation, we compare the synthesis flow optimization performance of CB-EVO with SOTA methods on various open-source designs. Since the baselines were originally evaluated on different benchmark suites, we present the results in two separate parts: for the comparisons in Section 5.2, we use the VTR benchmark suite [22], detailed in Table 3; for additional comparisons, we employ the EPFL benchmark suite [31], with statistics provided in Table 4.

Methods. We compare our CB-EVO with recent SOTA baselines [10, 14, 18, 23] and the conference version of our work [32]. These methods fall into two categories: those using lightweight bandit algorithms and those employing classical RL with complex neural network training. Experiments are performed within the action space detailed in Section 3.1 to minimize the number of AIG nodes and 6-input LUTs.

Abbreviations. For clarity, the key abbreviations used in the subsequent parts are as follows: "#LUT" for the number of 6-input LUTs, "Lvls" for circuit hierarchy levels, and "τ" for the algorithm's runtime. "ADP" is defined as the product of the LUT count and circuit levels. The modifiers $\bar{}$ (average) and $\hat{}$ (minimum) are applicable to both "#LUT" and "Lvls". Additionally, we utilize the

geometric mean to evaluate results, labeled as "Average" in the table, to minimize skew from extreme values across different design sizes.

5.2 Comparison with Bandit-based Method

Our approach is evaluated on the VTR benchmark suite against FlowTune [18], AlphaSyn [14], and CBTune [32]. We focus on two primary metrics: the number of AIG nodes after logic minimization and the number of LUTs after FPGA technology mapping.

To ensure a fair comparison, all methods are configured to use 42 transformations per synthesis flow ($L_2 = 42$), with each transformation selected from the 7 candidates in $\mathcal{A}$. For FlowTune, we set the "stages:iterations" parameter to 3 : 20 with 2-repetition, yielding 14 transformations per stage and a total synthesis flow length of 42. To provide a more comprehensive performance analysis, we run FlowTune with different random seeds to generate a broader set of results. We also include AlphaSyn [14], an MCTS-based framework that employs a domain-specific search strategy and leverages an online-trained neural network with a self-syn data collection mechanism. AlphaSyn is included because its SynUCT search strategy is closely related to the UCB criterion commonly used in bandit algorithms. For AlphaSyn, we set the number of training rounds to 3 and perform 20 self-syn runs for data collection in each round. PQnet training is not used; instead, we adopt uniform sampling based on historical statistics (Q+R), rather than relying on the neural network output, to achieve higher efficiency. In addition, we evaluate a Greedy baseline, which at each decision step exhaustively applies all candidate transformations $a \in \mathcal{A}$ to the current AIG and selects the one that yields the best immediate improvement in the target metric. To illustrate the range of possible outcomes, we generate 50,000 random synthesis flows to estimate the distribution of optimization results. We further include the classic "Resyn" [1] script from ABC as another baseline. The transformations in the Resyn script are all included in our candidate set $\mathcal{A}$. Specifically, we construct a sequence consisting of three `resyn2` commands (each containing 10 transformations) and two `resyn` commands (each containing 6 transformations), matching the total length of 42, and insert the same supplementary subsequence as FlowTune for consistency.

For our proposed methods, CBTune is configured to perform T contextual bandit iterations at each step, where T is dynamically scheduled based on the iteration budget and the current step i (with $T_{orig} = 15$, $T_{min} = 8$, and $decay\_rate = 0.4$). Multiple search depths $l \in 3, 4, 5$ are used, with 3 search times ($m = 3$) at each depth, to enable accurate extraction of the contextual feature x^l. In CB-EVO, we first use the bandit algorithm to explore the initial one-third of the synthesis sequence, and then employ an EA to efficiently search the remaining part of the sequence. This bandit configuration is consistent with that of CBTune. During the evolutionary phase, after each generation sequence is produced, we intermittently insert a `&choice_store` command. Once three such stores have been accumulated, a `&choice_compute` command is executed to further the optimization process. It is important to note that the CHOICE commands are not included in the total count of 42 transformations, as they do not perform additional optimization operations on the circuit.

AIG Node Count Optimization. The experimental results for logic optimization are shown in Figure 9, which plots the AIG node count (x-axis) against the corresponding AIG logic depth (y-axis). Our primary optimization objective is to minimize the AIG node count, while also aiming to reduce logic depth as much as possible. Thus, points closer to the lower left corner indicate better results. For reference, the initial node and logic depth of each benchmark are listed in Table 3. Note that we exclude the initial nodes' corresponding data points from the plots, as they are significantly higher than all baselines' worst results (17.8% higher in node count on average), which cluster at the

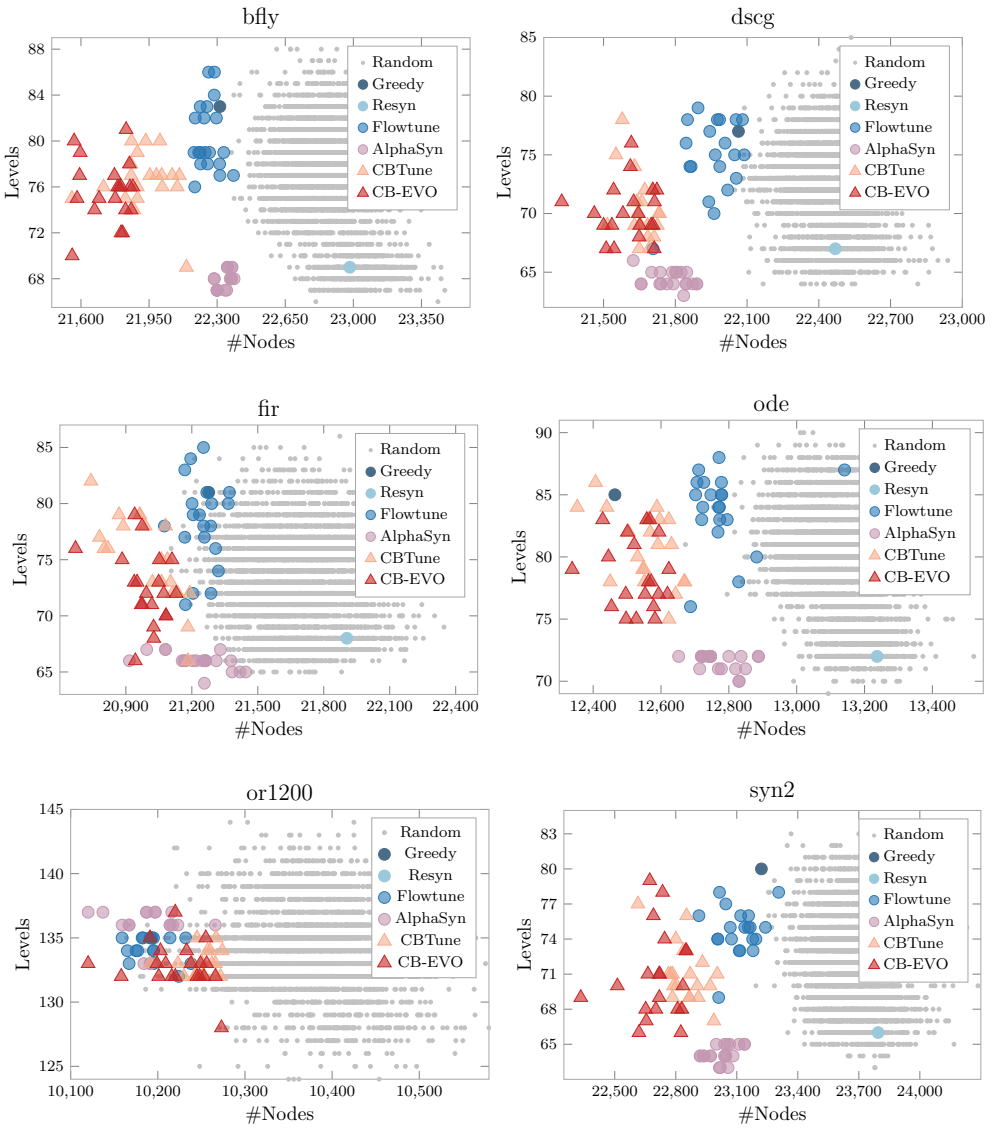


Fig. 9. Comparison of CB-EVO and baselines [14, 18, 32] on AIG node optimization. The main objective is to minimize AIG node count (x-axis); logic depth (y-axis) is for reference. Points closer to the lower left are preferred.

extreme upper-right corner and make the charts difficult to interpret. For each method (AlphaSyn, FlowTune, CBTune, and CB-EVO), we conduct 20 independent runs, with each point in the figure representing the optimization result of one synthesized flow. Greedy always produces a single deterministic result, so only one point is shown; similarly, Resyn is a fixed script and also yields a single point. To ensure a fair comparison, CB-EVO and all baseline methods follow FlowTune's implementation by inserting the supplementary subsequence "`ifraig; dch -f;`" after every 14 transformations.

Table 5. Comparison of LUT Count Optimization Results on VTR 8.0 Benchmarks

Benchmark	Greedy	Resyn	Flowtune [18]		AlphaSyn [14]		CBTune [32]		CB-EVO	
	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs	#LUTs
bfly	8,269	8,567	8,073	8,121.65	7,944	8,007.47	7,962	8,086.03	**7,882**	**8,002.81**
dscg	8,313	8,641	8,107	8,175.83	**7,887**	8,044.60	7,981	8,119.84	7,959	**8,074.69**
fir	8,385	8,297	7,952	7,992.63	7,894	7,949.40	7,820	7,977.38	**7,627**	**7,900.81**
ode	5,316	5,465	5,037	5,058.65	4,948	5,037.91	4,920	5,046.71	**4,917**	**5,021.31**
or1200	2,748	2,789	2,726	2,733.32	2,707	2,728.95	2,731	2,754.07	**2,700**	**2,718.71**
syn2	8,669	8,852	8,341	8,398.14	8,140	8,256.47	8,234	8,360.53	**8,071**	**8,248.06**
Average	6,464.69	6,602.86	6,250.02	6,284.86	6,146.55	6,223.90	6,166.39	6,271.82	**6,094.99**	**6,212.44**
Ratio	1.029	1.051	0.994	1.000	0.978	0.990	0.981	0.998	**0.970**	**0.988**

Table 6. Comparison of Runtime (in Minutes, τ) for LUT Count Optimization on VTR 8.0 Benchmarks

Benchmark	Flowtune [18]	AlphaSyn [14]			CBTune [32]	CB-EVO
	$\tau(m)$	self-syn(m)	infer(m)	$\tau(m)$	$\tau(m)$	$\tau(m)$
bfly	76.47	24.99	9.34	34.32	29.63	**23.00**
dscg	77.15	19.08	7.28	26.36	30.44	**22.51**
fir	74.23	21.35	8.54	29.89	27.60	**21.62**
ode	34.83	18.68	6.26	24.94	17.32	**12.92**
or1200	20.08	12.50	4.63	17.13	15.62	**9.66**
syn2	81.33	28.83	9.16	37.98	31.67	**23.08**
Average	54.04	20.23	7.32	27.57	24.44	**17.84**
Ratio	1.000	-	-	0.510	0.452	**0.330**

The results demonstrate that the synthesis flows generated by CB-EVO achieve the most effective logic optimization overall. Compared to Greedy, Resyn, FlowTune, AlphaSyn, and CBTune, CB-EVO achieves average reductions in AIG node count of approximately 1.71%, 4.64%, 1.42%, 1.27%, and 0.36%, respectively. In terms of the best AIG node count observed (i.e., the lowest value among all runs), CB-EVO consistently outperforms the other methods—especially on designs such as *dscg* and *syn2*—while also maintaining reasonable logic depth, frequently achieving results closest to the lower left corner.

LUT Count Optimization. For reducing the 6-input LUT count after FPGA technology mapping, results are shown in Table 5 and Table 6. Both CB-EVO and other baseline methods adopt 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. To ensure reliability, except for Greedy and Resyn, which produce a single deterministic result, Flowtune, AlphaSyn, CBTune, and CB-EVO are each run 20 times. Table 5 reports both the best and average outcomes across these runs, showing that CB-EVO achieves the greatest reduction in LUT count. Compared to the average results of Flowtune, AlphaSyn, and CBTune, CB-EVO improves the average LUT count by 0.97%, 0.21%, and 1.15%, respectively. CB-EVO also delivers the best result for all cases except *dscg*.

As shown in Table 6, CB-EVO's runtime is 3.03 times faster than Flowtune and 1.55 times faster than AlphaSyn. For AlphaSyn, the table details runtime as "self-syn" (data collection over three rounds, sampling only, no PQnet training in the non-NN setting), "infer" (inference and search time), and their sum (τ) for comparison. The introduction of the EVO model enables CB-EVO to achieve a 1.37x speedup in average runtime compared to CBTune. Note that the runtimes of "Greedy" and "Resyn" are not reported, since "Greedy" performs parallel local evaluations and "Resyn" executes a fixed deterministic flow, both without model-driven search and not reflective of the

Table 7. Comparison of Average LUT Count (#LUTs) for LUT Optimization on EPFL Benchmarks

Benchmark	Greedy	Resyn	DRiLLS [10]	RL4LS* [23]	AlphaSyn [14]	CBTune [32]	CB-EVO
max	697	744	694	687.80	692	684.25	**681.70**
adder	**244**	249	**244**	**244**	**244**	**244**	**244**
cavlc	115	116	112.20	111.30	109.85	111	**108.80**
ctrl	28	29	28	28	28	28	**27.90**
int2float	46	46	42.60	42.30	40.25	40	**39.80**
router	67	72	70.10	69.50	66.85	68.11	**65.15**
priority	146	186	**133.40**	142.90	138.05	138.86	134.95
i2c	291	288	292.10	289.32	301.25	283.11	**280.90**
sin	1,451	1,473	1,441.50	**1,438**	1,447.65	1,441.67	1,439.55
square	3,898	3,915	3,889.40	3,889	**3,875**	3,882.11	3,879.55
sqrt	4,807	4,890	4,708	4,685.30	4,663.20	4,607	**4,584.75**
log2	7,660	7,738	7,583.60	7,580.10	7,583	**7,580**	**7,580**
multiplier	5,688	5,713	5,678	**5,672**	5,674	5,679.75	5,674
voter	1,904	1,912	1,834.70	1,678.10	**1,550**	1,682.25	1,593.35
div	4,205	8,185	7,944.40	7,807.10	8,161.75	4,180.91	**4,142.84**
mem_ctrl	**10,144**	11,321	10,527.60	10,309.70	10,194.35	10,242.57	10,149.95
Average	732.69	792.93	753.49	748.34	741.96	712.83	**704.26**
Ratio	0.972	1.052	1.000	0.993	0.985	0.946	**0.935**

*_Last10_ in RL-PPO-Pruned [23].

intended runtime comparison among learning-based methods. Overall, these results demonstrate that CB-EVO is both effective in LUT count optimization and efficient in runtime.

5.3 Comparison with NN-enhanced Classical RL Method

We further compare our approach with two other RL-based SOTA works: DRiLLS [10] and RL4LS [23], targeted at reducing the LUT count after FPGA mapping. In contrast to our online-learning bandit method, these works rely on classical RL techniques and necessitate pre-trained neural network models to guide their decisions for the synthesis flow. Our experiments use the EPFL benchmark suite with a synthesis flow length of $L_2 = 25$, aligned with the baseline settings mentioned in their article. We incorporate the priority cuts mapper [33] in ABC, specifically employing the command "`if -a -K 6;`" to optimize mapping for LUT count and enhance area efficiency. For reference, we also report results from Greedy, Resyn, and AlphaSyn [14]. The settings for CB-EVO and CBTune are consistent with those described in Section 5.2.

Table 7 presents the comparison of the LUT count optimization results between CB-EVO and other methods, while the total runtimes for all approaches are listed in Table 8. Note that, except for the "Greedy" and "Resyn" methods, which can only report a single result, all other methods are executed twenty times to report the final synthesis flow optimization results, with the average target values presented. In Table 8, the reported runtime (τ) is the average total runtime, which includes the time required for collecting and training data within the framework, as well as the time for making decisions and executing synthesis flow optimizations. For AlphaSyn, we specifically provide a detailed breakdown of the runtime: the time for "self-syn" data collection (3 rounds; sampling only without PQnet training in the without NN setting), the time for "infer" (inference and search), and their sum (τ) for comparison. As in the previous section, the runtimes of "Greedy" and "Resyn" are omitted for consistency with the earlier comparison.

The results demonstrate that CB-EVO is highly effective in optimizing LUT count, achieving a score of 0.935 relative to DRiLLS (normalized to 1.000). This corresponds to a 5.9% improvement over RL4LS, a 5.1% improvement over AlphaSyn, and a 1.2% improvement compared to CBTune.

Table 8. Comparison of Total Runtime (τ, Minutes) for LUT Optimization on EPFL Benchmarks

Benchmark	DRiLLS [10]	RL4LS* [23]	AlphaSyn [14]			CBTune [32]	CB-EVO
	$\tau(m)$	$\tau(m)$	self-syn(m)	infer(m)	$\tau(m)$	$\tau(m)$	$\tau(m)$
max	32.58	54.34	15.54	0.32	15.87	6.01	**3.77**
adder	24.05	10.05	12.16	0.21	12.37	5.97	**1.82**
cavlc	26.02	3.22	7.43	0.17	7.60	2.37	**2.03**
ctrl	24.25	2.85	8.54	0.25	8.79	**0.59**	1.56
int2float	21.70	2.81	7.07	0.19	7.27	2.76	**1.63**
router	22.01	3.07	6.31	0.16	6.47	2.32	**1.78**
priority	23.32	5.90	7.29	0.14	7.43	3.41	**1.96**
i2c	25.17	7.55	8.11	0.19	8.30	3.61	**2.15**
sin	51.15	20.10	11.68	0.25	11.93	9.71	**7.00**
square	130.00	72.88	19.92	1.34	21.26	25.99	**22.34**
sqrt	147.64	196.15	33.71	2.12	35.83	36.51	**32.71**
log2	198.60	125.28	50.36	4.45	54.80	41.27	**39.88**
multiplier	180.84	187.81	40.21	3.13	43.34	29.08	**27.38**
voter	84.43	330.48	19.80	0.60	20.40	11.46	**11.28**
div	259.75	482.00	48.82	3.94	52.76	25.58	**24.61**
mem_ctrl	229.33	1985.84	45.28	1.60	46.88	45.81	**28.44**
Average	59.49	34.55	16.18	0.56	16.74	8.38	**6.56**
Ratio	1.000	0.581	-	-	0.281	0.141	**0.110**

**Last10* in RL-PPO-Pruned [23].

Furthermore, CB-EVO's runtime is only 0.110 times that of DRiLLS, making it 5.3% faster than RL4LS, 2.6% faster than AlphaSyn, and 1.3% faster than CBTune. On average, CB-EVO processes each design in approximately 6.56 minutes, reflecting the lowest average runtime among all methods. These substantial gains in both performance and efficiency highlight CB-EVO's strong competitiveness for applications that demand both speed and synthesis quality.

5.4 Ablation Study on the CHOICE Command

In our ablation study, detailed in Table 9, we assess the impact of the CHOICE command on the algorithm's performance by comparing optimization results with and without its inclusion. This experiment also utilizes the EPFL benchmark suite and performs LUT mapping with the optimization command "`if -a -K 6;`", aimed at minimizing the number of LUTs.

The analysis focuses on the geometric mean of average and best performance metrics across all benchmark cases. The results demonstrate a noticeable improvement when the CHOICE command is incorporated. Specifically, the average performance with the command is 704.26, which is approximately 0.30% better than the 706.39 observed without it. More significantly, the best performance also improves with the command, recording a geometric mean of 693.13 compared to 694.12 without the command, showing an improvement of about 0.14%. Although the magnitude of improvement is modest, it highlights the command's value in augmenting the effectiveness and robustness of the algorithm across various design sizes and configurations. The strategic use of the CHOICE command expands and perturbs the solution space, facilitating the escape from local optima and providing further directions for optimization.

5.5 Hyperparameter Sensitivity Analysis

In this subsection, we analyze the sensitivity of several heuristic hyperparameters in CB-EVO, quantifying their effects on solution quality and runtime. Specifically, we study: (i) with a fixed total sequence length L_2, how the contextual bandit phase length (L_1) affects performance, where the evolutionary phase covers the remaining $L_2 - L_1$ steps; (ii) for the arm's long-term payoff (x_a^l),

Table 9. Comparative Performance Analysis of CB-EVO with and without Choice Command

Benchmark	w/o Choice (Avg.)		w/ Choice (Avg.)		w/o Choice (Best)		w/ Choice (Best)	
	#LUTs	Lvls	#LUTs	Lvls	#LUTs	Lvls	#LUTs	Lvls
max	683.10	108	**681.70**	107.95	680	112	680	107
adder	**244**	116	**244**	116	244	116	244	116
cavlc	109.29	6.43	**108.80**	6.90	107	7	107	7
ctrl	27.95	2	**27.90**	2	27	2	27	2
int2float	39.95	5	**39.80**	5	39	5	38	5
router	65.67	8.33	**65.15**	8.40	62	9	64	8
priority	135.90	26.15	**134.95**	26.85	132	28	129	28
i2c	281.35	7.95	**280.90**	8.10	274	7	275	7
sin	1,441.95	75.40	**1,439.55**	74.60	1,436	75	1,435	75
square	3,879.75	121.10	**3,879.55**	121.10	3,875	121	3,875	121
sqrt	4,585.45	1,829.20	**4,584.75**	1,829.30	4,546	1,810	4,546	1,810
log2	7,583.63	159.13	**7,580.90**	159.30	7,580	159	7,580	159
multiplier	5,674.75	126	**5,674**	126	5,674	126	5,674	126
voter	1,608.15	18.45	**1,593.35**	19.35	1,539	18	1,539	18
div	4,158.45	2,061.50	**4,142.84**	2,058.58	4,108	2,042	4,112	2,049
mem_ctrl	10,192.35	47.15	**10,149.95**	47.10	10,071	44	9,974	44
Average	706.39	48.16	**704.26**	37.66	694.12	48.22	693.13	47.74

how the number of search times (m) affects decision quality and efficiency; (iii) how the maximum number of transformations appended per evolutionary offspring generation (M) influences convergence behavior and overall performance, and (iv) the sensitivity of synthesis quality and runtime to the return-back threshold. We conduct our analysis using the "*mem_ctrl*" benchmark from the EPFL suite, with LUT count minimization as the optimization objective. Experimental settings follow Section 5.3 with fixed total sequence length $L_2 = 25$.

CMAB Model's Decision Horizon (L_1). Figure 10(a) analyzes the CMAB model's performance across different decision lengths L_1 with fixed total sequence length $L_2 = 25$, where the evolutionary model handles the remaining $L_2 - L_1$ steps. Results show that increasing L_1 generally improves synthesis quality (LUT count) but increases runtime. This demonstrates that selecting L_1 entails a quality–runtime tradeoff that must be tuned to design constraints.

Search Times (m). Figure 10(b) analyzes the CMAB model's performance across different search times m for the context generator to generate the long-term payoff x^l. Results show that increasing m generally improves synthesis quality (LUT count) but increases runtime to some extent. Hence, m should be chosen to balance synthesis quality gains against computational cost.

Maximum Transformations per EVO Generation (M). Figure 10(c) shows that increasing M expands the evolutionary search and initially improves synthesis quality. The LUT count decreases markedly from M=1 to M=5, reaching its minimum at M=7, while runtime rises almost linearly with M. Beyond this point, further enlargement causes slight fluctuations without clear gains, indicating a saturation of search benefit. Moderate values (M=5-7) mark the region where quality improvement and computational cost become well balanced.

Return-Back Mechanism Threshold. Figure 10(d) analyzes performance across return-back thresholds ranging from 0.0 to 3.0. In general, smaller thresholds lead to more frequent return-back operations and better synthesis quality, but at the cost of longer runtime, whereas larger thresholds reduce runtime but may miss beneficial corrections. Therefore, careful threshold selection is essential to balance optimization quality and computational efficiency.

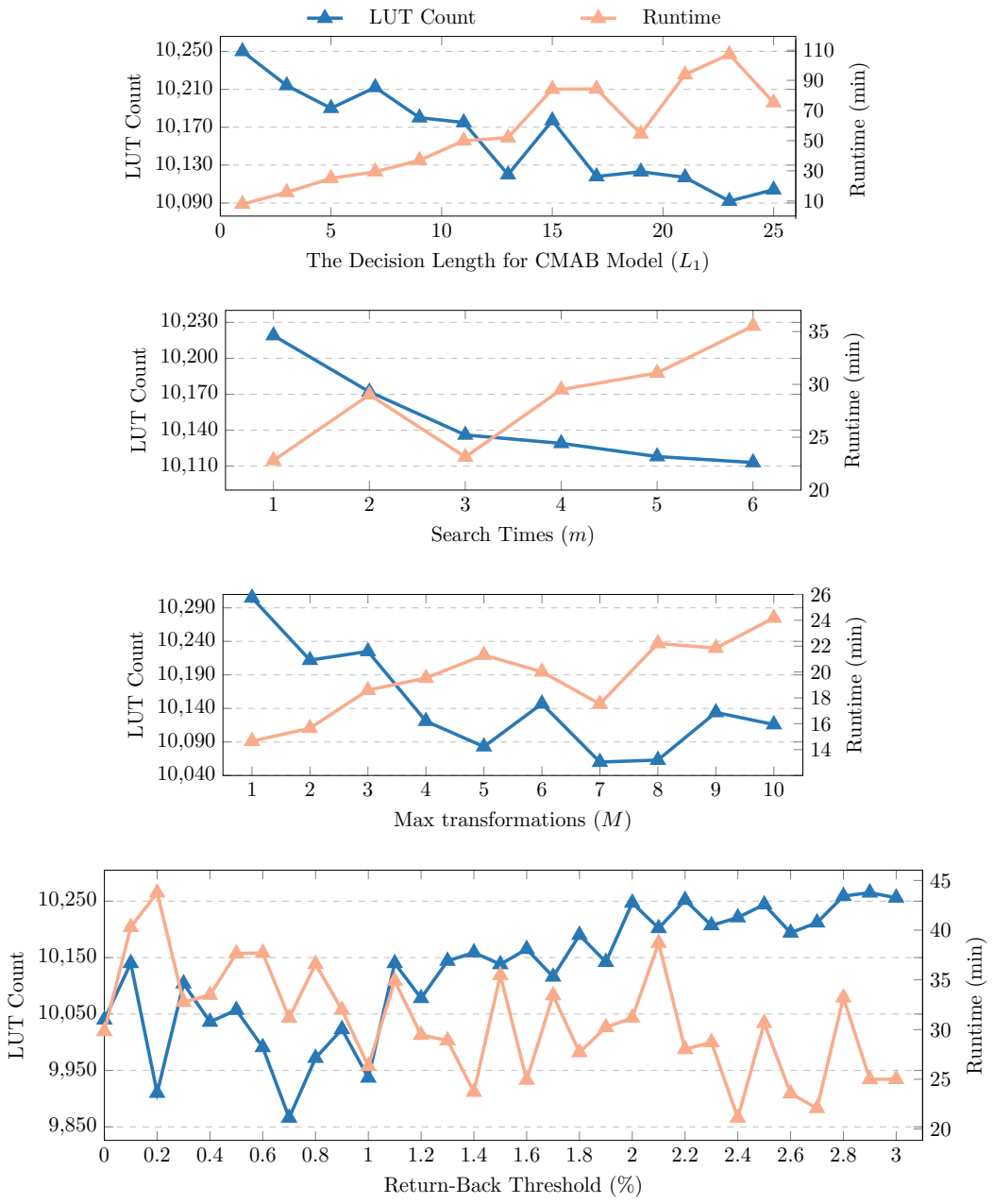


Fig. 10. Sensitivity of key hyperparameters: (a) bandit phase length L_1, (b) search repetitions m, (c) maximum transformations per EVO generation (M), and (d) return-back threshold.

5.6 Comparative Analysis of Optimization Objectives

In practical applications, certain use cases prioritize reducing the number of LUTs to lower costs and power consumption, while others may focus on decreasing the number of logic levels to enhance operational speed. Our experiment provides a deeper understanding of the algorithm's

Table 10. Comparative Performance Analysis of CB-EVO with Different Objectives

Benchmark	Target #LUTs			Target Lvls			Target ADP		
	#LUTs	Lvls	ADP	#LUTs	Lvls	ADP	#LUTs	Lvls	ADP
max	**681.70**	108	7.36E+04	746.25	**65.40**	4.88E+04	734.65	65.90	**4.84E+04**
adder	**244**	**116**	**2.83E+04**	244	**116**	**2.83E+04**	244	**116**	**2.83E+04**
cavlc	**108.80**	6.90	7.51E+02	116.43	**5**	5.82E+02	111.60	**5**	**5.58E+02**
ctrl	**27.90**	**2**	**5.58E+01**	28.57	**2**	5.71E+01	27.95	**2**	5.59E+01
int2float	**39.80**	5	**1.99E+02**	43.24	**4.90**	2.12E+02	40.10	4.95	**1.99E+02**
router	**65.15**	8.40	5.47E+02	71.52	7.29	5.21E+02	66.38	**7.14**	**4.74E+02**
priority	**134.95**	26.85	3.62E+03	153.75	**20.58**	3.16E+03	142.10	20.75	**2.95E+03**
i2c	**280.90**	8.10	2.28E+03	331.65	**5.85**	1.94E+03	296.65	5.95	**1.77E+03**
sin	**1,439.55**	74.60	1.07E+05	1,465.08	70.83	1.04E+05	1,462.60	**70.45**	**1.03E+05**
square	**3,879.55**	121.10	4.70E+05	3,896.65	121.05	4.72E+05	3,879.60	**121**	**4.69E+05**
sqrt	**4,584.75**	1,829.30	**8.39E+06**	4,589.90	**1,828.20**	**8.39E+06**	4,585.29	1,829.18	**8.39E+06**
log2	**7,580.90**	159.30	1.21E+06	7,679.92	**152.92**	**1.17E+06**	7,662.86	153.05	**1.17E+06**
multiplier	**5,674**	**126**	**7.15E+05**	5,709.15	**126**	7.19E+05	**5,674**	**126**	**7.15E+05**
voter	**1,593.35**	19.35	3.08E+04	1,813.62	18	3.26E+04	1,669.65	**17.70**	**2.96E+04**
div	**4,142.84**	2,058.58	**8.53E+06**	10,754.58	**2,041.42**	2.20E+07	4,145.15	2,057.25	**8.53E+06**
mem_ctrl	**10,149.95**	47.10	4.78E+05	10,417.62	**43.08**	4.49E+05	10,348.15	43.20	**4.47E+05**
Average	**704.26**	48.64	3.43E+04	788.44	**43.36**	3.42E+04	718.84	43.39	**3.12E+04**

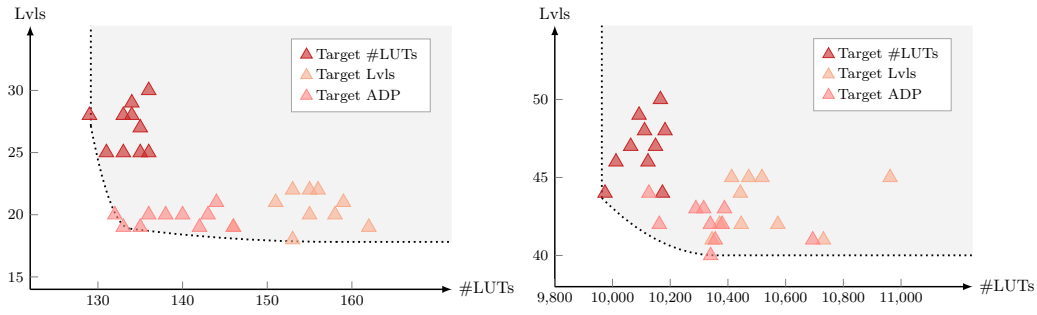
performance under various optimization objectives, allowing us to offer more precise and efficient design solutions tailored to specific needs.

We assess the impact of different optimization goals on the CB-EVO framework, specifically aiming to minimize LUT count, logic levels, and the Area-Delay Product (ADP). The results, as shown in Table 10, reveal clear tradeoffs between these objectives. For instance, targeting LUT count typically achieves the lowest LUT numbers, evident in benchmarks like "*max*" and "*cavlc*," where LUTs reduce to 681.70 and 108.80, respectively, significantly better than other targets, but occasionally at the cost of increased logic levels. When the focus shifts to minimizing logic levels, the average logic levels decrease (e.g., 43.36 vs. 48.64 when targeting LUT count), albeit slightly increasing the LUT count. Targeting ADP generally shows the best compromise, yielding the lowest ADP values, such as 558 and 55.9 for "*cavlc*" and "*ctrl*," respectively, indicating an effective balance between the two metrics. In Figure 11, we plot ten distinct points for the "*priority*" and "*mem_ctrl*" designs and connect the points on the boundary. The selected three objectives approximately follow a Pareto distribution, suggesting that while single-objective targeting can optimize specific aspects, the ADP approach offers a more balanced solution, potentially providing the best overall circuit performance for practical applications.

5.7 Additional Findings

During our experimental exploration, we identified several insights driven by empirical evidence. Although these are not included in the main evaluation previously for the sake of fairness, they show promising potential for improving the quality of synthesis flow generation results.

(1) Longer synthesis sequences generally lead to better optimization results. Therefore, instead of strictly limiting the sequence length (such as using L_2), allowing the process to continue based on a predefined time budget or until the optimization results converge could further improve PPA, as this permits the flow to keep expanding. As shown in Figure 12, we conducted experiments on several VTR8.0 benchmarks without any time constraints using CB-EVO. The x-axis represents the number of transformations in the sequence (i.e., the sequence length), while the y-axis shows the corresponding number of AIG nodes. It is evident that as the sequence length increases, the number of AIG nodes consistently decreases, albeit with minor



(a) *priority* (b) *mem_ctrl*

Fig. 11. Distribution of optimization objectives with Pareto-like characteristics.

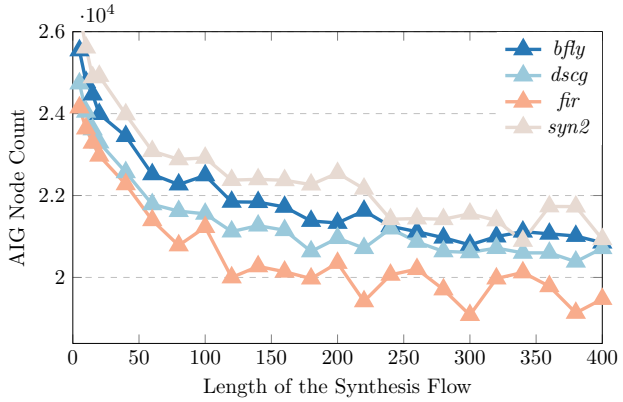


Fig. 12. AIG node count continues to drop as the synthesis sequence grows.

fluctuations. Notably, even when the number of transformations substantially exceeds the previously used limits of 42 and 25 (as referenced in Section 5.2 and Section 5.3, respectively), the optimization process does not fully converge. While the reduction in AIG node count tends to plateau and shows signs of convergence as the sequence becomes very long, these results indicate that allowing longer synthesis sequences can still yield incremental optimization benefits beyond the limits previously imposed.

(2) For LUT-based optimization tasks, incorporating ABC9-specific commands [34] such as `&extract`, `&dch`, `&synch2`, and `&dsdb` with tunable parameters, alongside replacing conventional mapping with `&if` and configuring cuts-per-node thresholds alongside LUT sizes, could unlock additional optimization gains. Post-mapping, leveraging don't-care-aware commands like `&mfs` may further reduce area. Notably, the recently introduced GIA manager in ABC9, which supports XOR and MUX nodes alongside standard AND nodes, enables dynamic switching between AIG and XAIG representations via `&get` and `&put` commands. This capability facilitates structural flexibility during optimization, potentially improving the QoRs by exploiting heterogeneous node types in the GIA framework.

In general, our CB-EVO 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.

6 Conclusion

In summary, this work presents CB-EVO, a novel framework that synergistically integrates contextual bandit tuning with the EA to automate the generation of high-quality synthesis flows, thereby enhancing the QoRs in circuit design. The CMAB model incorporates a context generator to facilitate informed decision-making and utilizes the Syn-LinUCB algorithm as its agent to iteratively evaluate and select the optimal arm. The EVO model complements the bandit tuning process by enabling faster exploration and extension of sequences. Additionally, we implement several optimization techniques, employing the "return-back" mechanism in the CMAB model to prevent the algorithm from falling into local optima and leveraging lossless synthesis to enhance the choice network, which creates more opportunities for optimization during the search process. Experimental results highlight CB-EVO's excellence in optimizing AIG nodes and 6-LUTs within an ideal runtime. The framework is open-source and available at GitHub.[1]

References

[1] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*. 24–40.

[2] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. 2018. Developing synthesis flows without human knowledge. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[3] Walter Lau Neto, Max Austin, Scott Temple, Luca Amaru, Xifan Tang, and Pierre-Emmanuel Gaillardon. 2019. LSOracle: A logic synthesis framework driven by artificial intelligence. In *Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

[4] Yuan Pu, Fangzhou Liu, Zhuolun He, Keren Zhu, Rongliang Fu, Ziyi Wang, Tsung-Yi Ho, and Bei Yu. 2025. HeLO: A heterogeneous logic optimization framework by hierarchical clustering and graph learning. In *Proceedings of the 2025 International Symposium on Physical Design*. 116–124.

[5] Nan Wu, Jiwon Lee, Yuan Xie, and Cong Hao. 2022. Lostin: Logic optimization via spatio-temporal information with hybrid graph models. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 11–18.

[6] Haisheng Zheng, Zhuolun He, Fangzhou Liu, Zehua Pei, and Bei Yu. 2024. LSTP: A logic synthesis timing predictor. In *Proceedings of the 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 728–733.

[7] Faezeh Faez, Raika Karimi, Yingxue Zhang, Xing Li, Lei Chen, Mingxuan Yuan, and Mahdi Biparva. 2025. MTLSO: A multi-task learning approach for logic synthesis optimization. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. 72–78.

[8] Animesh Basak Chowdhury, Benjamin Tan, Ryan Carey, Tushit Jain, Ramesh Karri, and Siddharth Garg. 2022. Bulls-Eye: Active few-shot learning guided logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 8 (2022), 2580–2590.

[9] Chenhui Deng, Zichao Yue, Cunxi Yu, Gokce Sarar, Ryan Carey, Rajeev Jain, and Zhiru Zhang. 2024. Less is more: Hop-wise graph attention for scalable and generalizable learning on circuits. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[10] Abdelrahman Hosny, Soheil Hashemi, Mohamed Shalan, and Sherief Reda. 2020. DRiLLS: Deep reinforcement learning for logic synthesis. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 581–586.

[11] Keren Zhu, Mingjie Liu, Hao Chen, Zheng Zhao, and David Z. Pan. 2020. Exploring logic optimizations with reinforcement learning and graph convolutional network. In *Proceedings of the ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*. 145–150.

[12] Yu Qian, Xuegong Zhou, Hao Zhou, and Lingli Wang. 2024. An efficient reinforcement learning based framework for exploring logic synthesis. *ACM Transactions on Design Automation of Electronic Systems* 29, 2 (2024), 1–33.

[13] Animesh Basak Chowdhury, Marco Romanelli, Benjamin Tan, Ramesh Karri, and Siddharth Garg. 2024. Retrieval-guided reinforcement learning for boolean circuit minimization. *arXiv preprint arXiv:2401.12205*.

[14] Zehua Pei, Fangzhou Liu, Zhuolun He, Guojin Chen, Haisheng Zheng, Keren Zhu, and Bei Yu. 2023. AlphaSyn: Logic synthesis optimization with efficient Monte Carlo tree search. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[1] https://github.com/sallyliu921/CB-EVO.git

[15] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. 2021. Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv preprint arXiv:2110.11292*.

[16] Antoine Grosnit, Cedric Malherbe, Rasul Tutunov, Xingchen Wan, Jun Wang, and Haitham Bou Ammar. 2022. Boils: Bayesian optimisation for logic synthesis. In *Proceedings of the 2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1193–1196.

[17] Cunxi Yu. 2020. Flowtune: Practical multi-armed bandits in boolean optimization. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[18] Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. 2022. FlowTune: End-to-end automatic logic optimization exploration via domain-specific multiarmed bandit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 6 (2022), 1912–1925.

[19] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd Annual Design Automation Conference*. 532–535.

[20] Alan Mishchenko, Robert Brayton, Stephen Jang, and Victor Kravets. 2011. Delay optimization using SOP balancing. In *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 375–382.

[21] Wassim Jouini, Damien Ernst, Christophe Moy, and Jacques Palicot. 2010. Upper confidence bound based decision making strategies and dynamic spectrum access. In *Proceedings of the IEEE International Conference on Communications (ICC)*. 1–5.

[22] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 7, 2 (2014), 1–30.

[23] Guanglei Zhou and Jason H. Anderson. 2023. Area-driven FPGA logic synthesis using reinforcement learning. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 159–165.

[24] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[25] Michael D. Hutton, Jonathan Rose, Jerry P. Grossman, and Derek G. Corneil. 1998. Characterization and parameterized generation of synthetic combinational benchmark circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17, 10 (1998), 985–996.

[26] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the Web Conference*. 661–670.

[27] Thomas J. Walsh, István Szita, Carlos Diuk, and Michael L. Littman. 2012. Exploring compact reinforcement-learning representations with linear regression. *arXiv preprint arXiv:1205.2606*.

[28] Agoston E. Eiben and James E. Smith. 2015. What is an evolutionary algorithm? In *Introduction to Evolutionary Computing*. Springer, 15–35.

[29] Satrajit Chatterjee, Alan Mishchenko, Robert K. Brayton, Xinning Wang, and Timothy Kam. 2006. Reducing structural bias in technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 12 (2006), 2894–2903.

[30] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. 2005. *FRAIGs: A Unifying Representation for Logic Synthesis and Verification*. Technical Report. ERL Technical Report.

[31] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *Proceedings of the IEEE/ACM International Workshop on Logic Synthesis*.

[32] Fangzhou Liu, Zehua Pei, Ziyang Yu, Haisheng Zheng, Zhuolun He, Tinghuan Chen, and Bei Yu. 2024. CBTune: Contextual bandit tuning for logic synthesis. In *Proceedings of the 2024 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1–6.

[33] Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, and Robert Brayton. 2007. Combinational and sequential mapping with priority cuts. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 354–361.

[34] Benjamin L. C. Barzen, Arya Reais-Parsi, Eddie Hung, Minwoo Kang, Alan Mishchenko, Jonathan W. Greene, and John Wawrzynek. 2023. Narrowing the synthesis gap: Academic fpga synthesis is catching up with the industry. In *Proceedings of the 2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1–6.

Received 25 February 2025; revised 30 October 2025; accepted 30 November 2025