End-to-end Automatic Logic Optimization Exploration via Domain-specific Multi-armed Bandit

Walter Lau Neto *IEEE Student Member*, Yingjie Li *IEEE Student Member*, Pierre-Emmanuel Gaillardon *IEEE Senior Member*, Cunxi Yu *IEEE Member*

***Abstract*—Design flows are the explicit combinations of design transformations, primarily involved in synthesis, placement and routing processes, to accomplish the design of Integrated Circuits (ICs) and System-on-Chip (SoC). Mostly, the flows are developed based on the knowledge of the experts. However, due to the large search space of design flows and the increasing design complexity, developing *Intellectual Property (IP)-specific* synthesis flows providing high Quality of Result (QoR) is extremely challenging.**

Recent years have seen increasing employment of decision intelligence in electronic design automation (EDA), which aims to reduce the manual efforts and boost the design closure process in modern toolflows. However, existing approaches either require a large number of labeled data and expensive training efforts, or are limited in practical EDA toolflow integration due to computation overhead. This paper presents a generic end-to-end sequential decision making framework *FlowTune* for synthesis tooflow optimization, with a novel high-performance domain-specific, multi-stage multi-armed bandit (MAB) approach. This framework addresses optimization problems on Boolean optimization problems such as a) And-Inv-Graphs (# nodes), b) Conjunction Normal Form (CNF) minimization (# clauses) for Boolean Satisfiability; logic synthesis and technology mapping problems such as c) post static timing analysis (STA) delay and area optimization for standard-cell technology mapping, and d) FPGA technology mapping for 6-in LUT architectures. Moreover, we demonstrate the high extnsibility and generalizability of the proposed domain-specific MAB approach with end-to-end FPGA design flow, evaluated at post-routing stage, with two different FPGA backend tools (OpenFPGA and VPR) and two different logic synthesis representations (AIGs and MIGs). FlowTune is fully integrated with ABC [1], Yosys [2], VTR [3], LSOracle [4], OpenFPGA [5], and industrial tools, and is released publicly. The experimental results conducted on various design stages in the flow all demonstrate that our framework outperforms both hand-crafted flows [1] and ML explored flows [6], [7] in quality of results, and is orders of magnitude faster compared to ML-based approaches.

I. Introduction

Electronic Design Automation (EDA) is composed of different steps, each relying on different abstract levels to handle the complex tasks involved in the design of modern Integrated Circuits (ICs) design. Given the diverse set of algorithms and options to tune within the EDA flow, designing an efficient flow is a challenging and crucial task. Indeed, while EDA vendors provide generic reference design flows, a well-designed flow that is design-aware can greatly improve the *Quality of Results* (QoR), as well as reduce the time-to-market by reducing the number of iterations to achieve design closure. A primary barrier to rapid hardware specialization is the lack of guarantee by existing FPGAs CAD tools on achieving design closure in an out-of-the-box fashion. Using these tools usually requires extensive manual effort to calibrate and configure a large set of design parameters and tool options to achieve a high QoR. Unfortunately, evaluating just one design point could be painfully time-consuming, as design steps such as PnR usually consume hours to days for large circuits. In order to accelerate hardware innovation, there is an urgent need to lower the design cost by significantly reducing the time required to obtain accurate QoR estimation and minimizing human supervision in the design tuning process.

Recent years have seen an increasing application of ML to accelerate the design process and reduce human engineering efforts and are believed to have great potential to address more critical challenges in both ASIC and FPGA designs. Specifically, there are three major directions in applying ML in design flow optimizations – *(1) Fast and accurate approximation via predictive modeling* – ML can be used as a statistical technique that mines domain-knowledge from historical and existing data to forecast future or unseen outcomes w.r.t to specific algorithmic or mathematical objectives. With the recent progress in advanced ML algorithms and neural architectures, ML can be used to construct generic and accurate approximations for given objectives, which can significantly boost the design process [8], [9], [6], [10], [11]. A well-calibrated ML predictive model can replace such heavy computations with a fast approximation. *(2) Flexible and versatile modeling* – Unlike traditional statistical data analysis methods, modern ML techniques provide a wide range of modeling options to cover the complex FPGA design processes. On one hand, ML offers various predictive formulations that are essential to cover a large number of FPGA design challenges, e.g., classification, clustering, regression, generative modeling, etc ([6], [10], [12], [13], [14]). On the other hand, modern variants of ML are able to handle versatile feature representations such as graphs, circuit imaging, functional behaviors, etc., and learn complex behaviors between those features and target metrics. *(3) Minimizing human supervision* – Leveraging ML in FPGA design minimizes human supervision in the design process in two directions. First, the traditional CAD tool R&D process heavily relies on expert knowledge in FPGA design and CAD algorithms, and most heuristics are empirically developed with tremendous experimental efforts ([15], [16], [17], [7]). On the other hand, autonomous exploration and learning systems such as reinforcement learning mechanisms can significantly accelerate the exploration efforts with an intelligent self-guided agent.

In this context, recent years have seen an increasing employment of machine learning (ML) techniques to enable autonomous design space exploration, reducing manual efforts and boosting design closure [18], [8], [19], [20], [21], [22], [6], [7], [23], [24] for ASICs [21], [22], [6], [7] and FPGAs [18], [8], [19], [25], [26]. These works have two main flavors: (i) focus on EDA tool parameters tuning, i.e., binary switches (e.g., remapping on/off), and multi-label switches; (ii) exploring the sequence of synthesis transformations in an iterative training-exploration fashion through Convolutional Neural Networks (CNNs) [6] and reinforcement learning [7]. Also, the different proposed approaches rely in both offline [21] and online datasets [18], [8] have also been proposed. Still, the approaches proposed so far have the following limitations:

- **Lacking theoretical guarantees**. While ML-based approaches could generate promising results, there is no theoretical guarantees in exploration bound and failure prediction.
- **Lacking domain knowledge of synthesis algorithms**. Leveraging domain knowledge of the implemented algorithms leads to better initialization and final convergence. However, they are considered as black-box implementations in existing work [19], [8], [6], [7].
- **Lacking flexibility**. Once the ML-model have been constructed, such exploration systems stuck with a given space (because input features are fixed) and are limited to specific QoR objective(s) [8], [6], [7].
- **System integration overhead**. There is significant runtime overhead while EDA tools communicate with machine learning frameworks (e.g., TensorFlow used in [6], [7], [27], [28]).
- **Limited end-to-end validations**. Existing works on logic optimization explorations are all evaluated at post-synthesis or post-mapping stages, without evaluating results at post physical design stage [7], [6], [16], [15], which can have significant impacts on the QoRs.

Therefore, to tackle the aforementioned flaws, in this work we present and thoroughly discuss previously presented machine learning techniques for logic synthesis, and propose an easily portable light-weight multi-armed bandit (MAB) approach for Boolean logic optimization, called *FlowTune*. To show the portability and effectiveness of our approach, we integrate and present results for FlowTune in different scenarios: STA-timing aware standard-cell (STD) technology mapping, FPGA technology mapping, and Conjunction Normal Form (CNF) for Boolean Satisfiability (SAT). ASIC results are given post-mapping by a commercial STA tool, whereas FPGA results are collected post Place-and-Route (PnR). To show the flexibility of our approach, we integrate with two different logic optimization mechanisms, And-Inv-Graph (AIG) [29] and Majority-Inv-Graph (MIG) [30], and various design toolflows, such as ABC+LSOracle+VTR 8.0 [4], [3], ABC+Yosys [2] with different back-ends such as Vivado, OpenFPGA [5], and Cadence Genus. Specifically, the main contributions of this work include:

- A novel domain-specific bandit algorithm for sequential decision-making by leveraging domain knowledge of DAG-aware synthesis algorithms, with a detailed domain-knowledge illustration example.
- The proposed framework has been implemented in ABC and integrated with multiple open-source design toolflows for both ASICs and FPGAs evaluations, which enables end-to-end experimental evaluations to post-PnR stages.
- The proposed domain-specific bandit approach enables flexible and efficient exploration for logic optimizations with comprehensive experimental demonstrations, including Boolean minimization (depth and logic count), post-mapping optimization (delay and area), and post-PnR optimization (timing slacks, logic area, and routing area). We believe this is the first work that demonstrates the effectiveness of ML-guided synthesis exploration with complete PnR evaluations, using two different FPGA backend tools (VPR and OpenFPGA).
- We demonstrate the effectiveness of the domain-specific MAB approach in two different logic synthesis DAG representations, i.e., AIG-based and MIG-based logic synthesis. This is believed to be the first work that addresses MIG synthesis flow exploration in end-to-end settings.
- FlowTune framework and its integration of multiple toolflows will be fully released publicly [1] after acceptance.

II. Background

This section presents useful notations for the reader, as well as reviews the search space while generating synthesis flows. We then present a motivational example on how delay and area may vary with synthesis flow, and how these gains are comparable w.r.t the state-of-the-art.

A. Notations and Search Space

Definition 1 *none*-repetition Synthesis Flow: *Given a set of unique synthesis transformations $\mathbb{S}$={p_0, p_1,..., p_n}, a synthesis flow $\mathbb{F}$ is a permutation of $p_i \in \mathbb{S}$ performed iteratively.*

Example 1: Let $\mathbb{S}$={p_0, p_1, p_2}. p_i are the transformations in the synthesis tools and can be processed independently. Then, there are totally six flows available:

$$
\begin{array}{ll}
F_0: p_0 \rightarrow p_1 \rightarrow p_2 & F_1: p_0 \rightarrow p_2 \rightarrow p_1 \\
F_2: p_1 \rightarrow p_0 \rightarrow p_2 & F_3: p_1 \rightarrow p_2 \rightarrow p_0 \\
F_4: p_2 \rightarrow p_0 \rightarrow p_1 & F_5: p_2 \rightarrow p_1 \rightarrow p_0
\end{array}
$$

Remark 1: *Let $f(n,1)$ be the upper bound number of possible flows, where $\mathbb{S}$ includes n elements, and each transformation appears only once, such that:*

$$f(n,1) = n! \tag{1}$$

The upper bound of $\mathbb{N}$ happens *iff* all elements in $\mathbb{S}$ can be processed independently. In practice, there could be some constraints to be satisfied for processing these transformations. In this case, $\mathbb{N}$ will be smaller than $n!$. For example, given a constraint that p_1 has to be processed before p_2, the available flows include only F_0, F_2, and F_3.

Definition 2 m-repetition Synthesis Flow (m≥2): *Given a set of unique synthesis transformations $\mathbb{S}$={p_0, p_1,...,p_n}, a*

[1]https://github.com/Yu-Utah/FlowTune

synthesis flow with m-repetition $\mathbb{F}_m$ is a permutation of $p_i \in \mathbb{S}_m$, where $\mathbb{S}_m$ contains m $\mathbb{S}$ sets.

Example 2: Let $\mathbb{S}$=$\{p_0,\ p_1\}$. Each p_i can be processed independently. For developing 2-repetition synthesis flows, $\mathbb{S}_2$=$\{p_0,\ p_1,\ p_0,\ p_1\}$. The available flows include:

$$F_0 : p_0 \to p_0 \to p_1 \to p_1 \quad F_1 : p_1 \to p_1 \to p_0 \to p_0$$
$$F_2 : p_0 \to p_1 \to p_0 \to p_1 \quad F_3 : p_1 \to p_0 \to p_1 \to p_0$$
$$F_4 : p_0 \to p_1 \to p_1 \to p_0 \quad F_5 : p_1 \to p_0 \to p_0 \to p_1$$

Remark 2: *Let $\mathbb{L}$ be the length of a synthesis flow. Given a m-repetition $\mathbb{F}_m$ with n transformations in $\mathbb{S}$, $\mathbb{L}$ = $n \times m$.*

The search space for m-repetition flows is a *multiset* permutation problem. Hence, a closed formula can be derived to describe the search space of m-repetition flows. The search space of m-repetition flows with n unique transformations is shown in Equation 2.

Remark 3: *Let function $f(n,m)$ be upper bound number of available m-repetition flows with n elements in S. $f(n,m)$ uniquely satisfies the following formula :*

$$f(n,m) = \frac{(n \cdot m)!}{(m!)^n} \tag{2}$$

With the multiset permutation concept, Yu et al. [6] generalized the formula to describe the search space for any type of m-repetition flows. Let n be the number of unique transformation, the M-repetition flows, M=$\{m_0, m_2, ..., m_{n-1}\}$, where m_i is the number of repetitions of the i^{th} transformation. In this case, the total number of possible flows is shown in Equation 2.

Remark 4: *Let function $f(n,\mathbb{L},\{m_0,m_2,...,m_{n-1}\})$ be upper bound number of flows with n elements in S. The i^{th} element in S appears m_i times. The function uniquely satisfies the following formula :*

$$f(n,\mathbb{L},\{m_0,m_2,...,m_{n-1}\}) = \frac{(m_0 + m_1 + \cdots m_{n-1})!}{(m_1!)(m_2!)\cdots(m_{n-1}!)} \tag{3}$$

Remark 5: *Let $\mathbb{L}$ be the length of the type of flows with upper bound $f(n,\mathbb{L},\{m_0,m_2,...,m_{n-1}\})$, $\mathbb{L}$=$\sum_{i=0}^{n} m_i$.*

Whereas the work in [6] constraints the search-space to m-repetitions flow, with the upper bound given by Equation 2, in this work we approach this problem in a more general way, where each repetition might have a different number of repetitions. In this case, the theoretical upper bound is given by Equation 3.

B. Motivating Example

We provide a motivating examples using the open-source logic synthesis framework ABC [1] shown in Figure 1. The experimental setup for this motivational example is as follows:

- $\mathbb{S}$=$\{$balance, restructure, rewrite, refactor, rewrite -z, refactor -z$\}$ (n=6); the elements in $\mathbb{S}$ are logic transformations in ABC[2] that can be processed independently.

[2]The names of these transformations are the same as the commands in ABC.

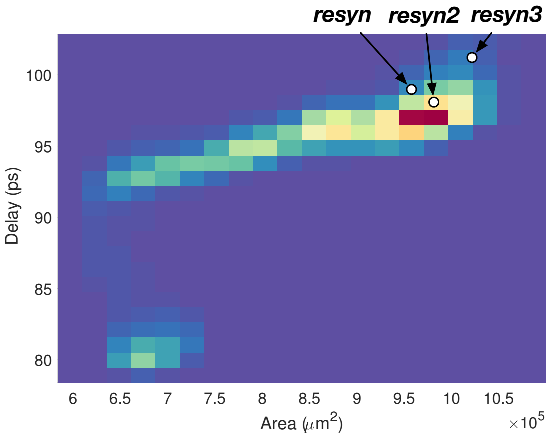


Fig. 1: Evaluation of three default flows, *resyn, resyn2* and *resyn3* using 128-bit AES design. The heatmap includes the QoR of the 50,000 random flows.

- 50,000 unique 4-repetition flows are generated by random permutations of $\mathbb{S}_4$ (m=4, n=6, $\mathbb{L}$=24).
- Input design: 128-bit Advanced Encryption Standard (AES) core[31].
- Delay and area of these flows are obtained after technology mapping using a 14nm standard-cell library.

The QoR distributions for the AES can be seen in Figure 1, and several important observations can be drawn from it, which highlight the main motivations of this work:

- Given the same set of synthesis transformations, the QoR is very different using different flows. For example, delay and area of AES design produced by the 50,000 flows have up to 40% and 90% difference, respectively.
- The search space of the synthesis flows is large. According to Remark 3, the total number of available 4-repetition flows with $n = 6$ independent synthesis transformations is more than 10^{16}. Discovering the high-quality synthesis flows with human-testing among the entire search space is unlikely to be achieved.

As it is possible to observe, we position the three most widely used default flows provided in ABC, including *resyn, resyn2* and *resyn3*, with respect to the QoR achieved with the random flows. Whereas *resyn* provides the best QoR among these three default flows, its result performs >10% worse than the best flows of the 50,000 random flows, in both delay and area. Specifically, there are more than 25,000 random flows that perform better than all the default flows. This means that for every two random permuted flows, one of them is likely to over-perform the expert-developed flows for this design. This, along with the large search space provides the main motivation for this work. It is also important to note that a given flow performs differently on different designs. For example, high-quality flows for our AES design could perform poorly on other designs [6].

```
1: procedure DAG-AWARE SYNTHESIS( )
2: $G(\mathcal{V},\mathcal{E}) \leftarrow$ circuit
3: for $v \in \mathcal{V}$ do
       if transformable($v$) then
           apply transformation to $v$
4:         update $G(\mathcal{V},\mathcal{E})$
       end
   end
5: end procedure
```

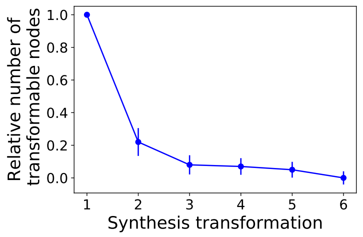


Fig. 2: Illustration of DAG-aware synthesis algorithm. And, the relative number of AIG nodes that are effectively executed in each transformation of the synthesis flows, using 100 random flows with six transformations used in [6], [7].

III. Approach

A. On the Impact of DAG-Aware Synthesis Algorithms

The most efficient algorithms that optimize the Boolean networks are directed acyclic graph (DAG) aware based Boolean synthesis algorithms [29], [2], which are widely used in both open-source tools [1], [2], [3] and industrial tools [32], [33], [34], [30]. Specifically, this work focuses on optimizing the synthesis flows that comprise DAG-aware synthesis algorithms and heuristics. Thus, to understand how effective each synthesis transformation (algorithm) is in the synthesis flows, we analyze the basic graph operations in DAG-aware algorithms. We can see that the number of actual transformed nodes (bracket 4 in pseudo-code Algorithm 2) represents the effectiveness of the algorithm for a given DAG (a circuit). Hence, to understand how effective each synthesis transformation (algorithm) is in the synthesis flows, we monitor the number of transformed nodes of all transformations using 100 random flows. The selected ABC synthesis transformations are the same six transformations as in [6], [7], namely $\mathbb{S}$={balance, restructure, rewrite, refactor, rewrite -z, refactor -z}.

The analysis results are shown in Figure 2, where the y-axis represents the relative number of transformed nodes of each transformation, and the x-axis shows the steps of the synthesis flows. For example, given a random *none*-repetition flow composed by the transformations available in $\mathbb{S}$, assume the first transformation is applied to ∼1,000 nodes in the original graph and the second transformation is applied to ∼200 nodes in the updated graph. In which case, the relative percentage of the first two transformations of this example is denoted as 1 and 0.2. Therefore, in Figure 2, the relative number of transformed nodes of all randoms flows start with 1. As there are 100 random permuted flows are used in this analysis, the error-bars are used to indicate the upper/lower bound of transformable nodes among all the random generated flows.

This experiment leads us to two main observations:

- 1) *Given a random permuted synthesis flow using DAG-aware transformations, only the first and the second transformations can effectively optimize the logic network.*

In Figure 2, we can see that for any random flow, the third to sixth transformations in the flow are applied to less than 10% nodes compared to the first transformation. In this particular example, we observe many cases that the fifth and sixth transformations do not detect any transformable node, regardless of the permutations.

- 2) *The optimization performance of synthesis flow is dominated by the first transformation since it dominates the transformable nodes for the rest flow.*

In other words, picking an ineffective transformation as the first one in a synthesis flow will likely result in an inefficient flow regardless of how the remaining optimization are permuted. This is analogous to solving a non-convex optimization problem with an initialization that always converges at a bad local optimum. The main reason is that the DAG-aware synthesis algorithms are mostly implemented based on structural search. The performance of such algorithms heavily relies on the structure of a given DAG. As indicated in the first observation, the structure of the graph will be dominated by the first transformation. Thus, we can say that the first transformation in the flow dominates the performance of synthesis flows. With these two observations in mind, we propose a domain-knowledge Multi-Armed Bandits (MAB) approach for automatic logic synthesis flow generation.

Example 1: An illustrative example that demonstrates the importance of the extracted algorithmic domain knowledge presented in Figure 2 is shown in Table I. Two DAG-aware synthesis transformations from ABC are selected to build two different flows F_0 and F_1 and are applied to design `bfly` from VTR [3] benchmark. We focus on comparing the transformed AIG nodes (#TNodes) of `rewrite` and the final number of AIG nodes. This is because `rewrite` performs a technology-independent rewriting algorithm of AIG, which offers the main reductions of #AIG in this example. We can see that **(1)** *The transformations at the early stage of the flows have more impacts than the transformations at the late stage.* For example, in both flows, first `rewrite` successfully applies to more than 7× #AIG than the second `rewrite`. **(2)** *The choice of early transformations has significant impacts on the performance of the flow.* While early transformations have more impacts in the Boolean network, the DAG structure could change dramatically at the early stage compared to the late stage. For example, while apply `balance` first, first `rewrite` in F_0 applies to 951 nodes. However, without without `balance`, `rewrite` observes 1764 nodes that can be rewritten (F_1). Similar results can be observed from the remaining two `rewrite`.

TABLE I: Example of the algorithmic domain knowledge presented in Figure 2 using ABC synthesis transformation *rewrite* (rw) and *balance* (b) with design bfly from VTR 8.0 [3] benchmark. Note that rw is technology-independent rewriting of the AIG which offers the main AIG reductions in F_0 and F_1. #TNodes = Number of AIG nodes transformed by the corresponding AIG transformation.

F_0	*b*	*rw*	*b*	*rw*	*b*	*rw*	Final #AIG
#TNodes	817	**951**	825	**169**	831	**64**	26339
F_1	*rw*	*b*	*rw*	*b*	*rw*	*b*	Final #AIG
#TNodes	**1764**	824	**290**	834	**90**	832	**26182**

B. Domain-specific MAB Formulation

In a multi-armed bandit problem, an online algorithm must choose from a set of strategies in a sequence of n trials to maximize the total payoff of the chosen strategies. Such problems assume that a fixed amount of resources must be allocated between available choices in a way that maximizes the expected gain of a given objective. These problems are the principal theoretical tool for modeling the exploration-exploitation trade-offs, which is inherent to sequential decision making under uncertainty. Multi-armed bandit can be described as a tuple of $< \mathcal{A}, \mathcal{R} >$, where:

- *$\mathcal{A}$ is a known set of available choices (arms).*
- *At each time step t, an action a_t is triggered by choosing with one of the choice $a_t \in \mathcal{A}$.*
- *$\mathcal{R}$ is a reward function and $\mathcal{R}^{a_t}$ is the reward at time step t with action a_t.*
- *The objective of bandit algorithm is to maximize $\sum_i^t \mathcal{R}^{a_t}$*

In a classic MAB sequential decision-making environment, the available decisions at time step t are considered as arms. For example, considering the synthesis flow exploration problem, the selected synthesis transformations will be the set of arms $\mathcal{A}$. Let $\mathcal{A}$ include eight unique transformations $\mathcal{A}$={`resub`, `rewrite`, `...`, `refactor`}. Let $\mathcal{R}$ be the number of AIG nodes that have been reduced by applying the transformations, such that the objective of the bandit algorithm is to maximize $\sum_i^t \mathcal{R}^{a_t}$, i.e., minimize the number of AIG nodes. Let F be a decision sequence, where F=$\{a_0, a_1, ..., a_n\}$, $a_t \in \mathcal{A}$. This decision sequence F is a synthesis flow. A brute-force approach to maximize the reward for finding the best flow F plays with each transformation (each arm) for enough rounds so that eventually it gets the true probability of reward. Unfortunately, such kind of brute-force approach which is not practical. In this context, the main idea behind the bandit algorithm is gathering enough information to make the best overall decisions. During exploitation, the best-known option is taken according to previous plays. Unknown options within the search space will be explored with the exploration phase to gather additional information to close the gap between estimated probability and the true probability of reward function. This procedure is similar to the reinforcement learning approach for synthesis flow exploration without using internal states, e.g., some snapshot about the current Boolean network statistics. Although the classic MAB sequential decision making has been widely applied, it needs to be re-thought to fit in logic synthesis flow exploration. In particular, according to the two observations in Section III-A:

- Since the first synthesis transformation dominates the flow, the first action in exploration will dominate the true reward distribution $\mathcal{R}^*$ and the exploitation reward distribution $\mathcal{R}$. In other words, the initialization of the bandit algorithm dominates the gap between $\mathcal{R}^*$ and $\mathcal{R}$.
- While considering each transformation as an arm, each action corresponds to applying one synthesis transformation to the logic graph. Unlike the classic MAB problem that $\mathcal{R}^*$ is fixed over time, $\mathcal{R}^*$ in synthesis flow exploration changes at each time step, as the logic graph is updated by the transformation.

Therefore, to gather the domain knowledge of synthesis algorithms discussed in Section III-A, we propose a novel MAB environment by re-defining the arms and actions, which fits the logic synthesis flow exploration problem. Thus, let $\mathbb{P}(\mathcal{X})$ be a random permutation function over a set of decisions $\mathcal{X}$. Let $\mathbb{P}(x\|\mathcal{X})$ be a random permutation function over the set $\mathcal{X}$, $x \in \mathcal{X}$, such that $\mathbb{P}(x_i\|\mathcal{X})$ is a random permutation with x_i always being the first element in the permutation, $\mathbb{P}(x_i\|\mathcal{X}) \in \mathbb{P}(\mathcal{X})$. We define $\mathbb{P}(x\|\mathcal{X})$ to be the arms in the MAB environment, such that $\mathcal{A}$={ $\mathbb{P}(x_0\|\mathcal{X})$, $\mathbb{P}(x_1\|\mathcal{X})$, ..., $\mathbb{P}(x_n\|\mathcal{X})$ }, where n is the number of available decisions in the exploration problem. Specifically, n corresponds to the number of available synthesis transformations. **Unlike using traditional MAB algorithms, an action a_t at time t is a sampled permutation from $\mathbb{P}(x_i\|\mathcal{X})$.** In other words, a_t is a *multiset* over the set $\mathcal{X}$. Let $Q(a_t)$ be the action value that is obtained by applying a_t to a given Boolean network at t time step, the reward is r_t

$$r_t = Q(a_t) - Q(a_{t-1}) \implies Q(\mathbb{P}(x_i\|\mathcal{X})) - Q(\mathbb{P}(x_j\|\mathcal{X}))$$

where the i^{th} arm is played at t time step and j^{th} arm is played at $t-1$ time step. Finally, we use upper confidence bound (UCB) bandit algorithm as the agent, such that a_t is chosen with estimated upper bound $U_t(a)$. The adopted upper bound is defined as follows:

$$a_t = \arg\max_{a \in \mathcal{A}} Q(a) + U_t(a), U_t(a) = \sqrt{\frac{log\ t}{2N_t(a)}}$$

The performance of a multi-armed bandit algorithm is often evaluated in terms of its regret, defined as the gap between the expected payoff of the algorithm and that of an optimal strategy. In this work, the number of regrets equals to the number of synthesis flows that have been evaluated in the synthesis tool. Using the UCB algorithm, an asymptotic logarithmic total regret L_t is achieved:

$$\lim_{t \to \infty} L_t = 2log\ t \sum \Delta_a$$

where Δ_a is the differences between arms in $\mathcal{A}$.

C. Improving Convergence with Multi-stage Bandit

While the approach described in the previous section focuses on optimistic initialization, we propose a multi-stage

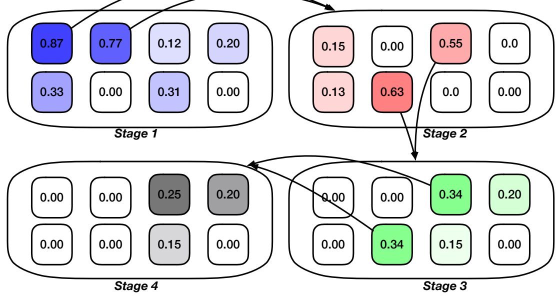


Fig. 3: Illustration of the proposed multi-stage bandit approach with four stages.

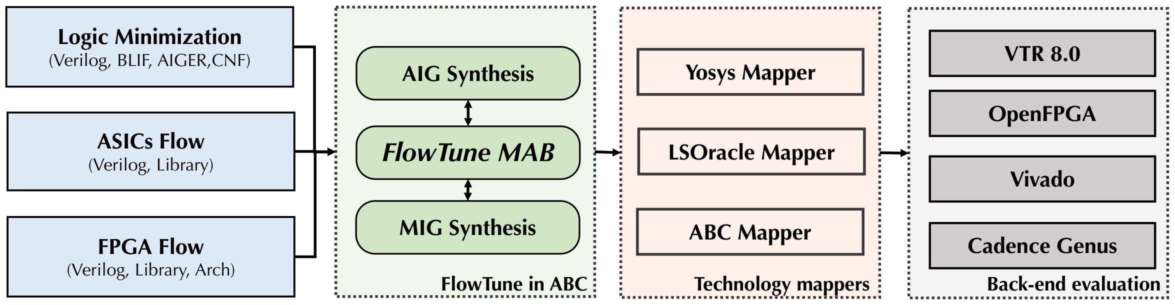


Fig. 4: Overview of the proposed end-to-end MAB synthesis system – Using ABC front-end, our system accepts technology mapped netlist, Boolean logic netlist, and LUT-netlist. The system is also integrated with VTR 8.0 and Yosys, which enable synthesis optimization for a large range of objectives for designing ASICs and FPGAs, and formal verification tools.

bandit to improve the convergence further. Based on the formulation in the last section, we can see that the single-stage approach can be applied to longer synthesis flows, with each transformation being repeated multiple times. However, increasing the length of the sequences leads to a significant increase in the number of explorations required to close the gap between the optimistic reward distribution $\mathcal{R}^*$ and exploitation reward distribution $\mathcal{R}$. Moreover, the optimistic reward distribution $\mathcal{R}^*$ changes as the synthesis transformations are applied to the logic circuit, since the graph structure is modified iteratively. Finally, although the synthesis transformations are less effective at late time steps, a fine-grain exploration can have great impact during the later stages of the EDA flows, e.g., technology mapping and gate sizing.

In this context, we introduce the proposed multi-stage bandit approach using a four-stage example shown in Figure 3. Each stage in the multi-stage approach uses the same domain-specific bandit algorithm described in Section III-A. Within each stage, the MAB algorithm is restricted to a fixed number of iterations m. Once the first stage is completed, the exploitation reward distribution $\mathcal{R}^1$ is updated (stage 1 in Figure 3), in which a higher rate indicates a higher chance of gaining reward by playing that arm. The best-explored synthesis flow after m iterations in the first stage will be applied to the input logic circuit, and the synthesized circuit will be the input circuit for the next stage. Instead of initializing a new MAB agent with uniform distribution, we initialize the second stage using the reward distribution of the top two arms in the first stage. For example, in Figure 3, $\mathcal{R}^1_{a_0}$ and $\mathcal{R}^1_{a_1}$ will be merged and used as the initial reward distribution for the second stage, where $\mathcal{R}^1_{a_0}, \mathcal{R}^1_{a_1} \in \mathcal{R}^1$. This procedure will continue until the s stages have been completed. As we can see, the total number of explorations is $s \cdot m$. In this work, we have explored five different options for (s, m), while maintaining the total number of iterations identical.

Example 2: We present an illustrative example of the aforementioned domain-specific MAB and the multi-stage bandit algorithm for exploring synthesis flows using the following ABC transformations: `rw`, `b`, `rf`, and `resub`. Let $\mathcal{X}$ be {`4×rw`, `4×b`, `4×rf`, `4×resub`}. Let the number of stages for exploration be four, such that $\mathcal{X}_{0,1,2,3}$ ={`rw`, `b`, `rf`, `resub`}. At first MAB stage, arms $\mathcal{A}_0$

$$\mathcal{A}_0 = \{\, \mathbb{P}(rw \| \mathcal{X}_0),\, \mathbb{P}(b \| \mathcal{X}_0),\, \mathbb{P}(rf \| \mathcal{X}_0),\, \mathbb{P}(resub \| \mathcal{X}_0)\, \} \tag{4}$$

are explored with the proposed MAB algorithm presented in Section III-A. The reward r_t is calculated based on the target objective, e.g., number of reduced AIG nodes compared to the existing best actions. Next, Stage 1 terminates after a fixed number of iterations, and returns the arm(s) with the highest reward value. For the sake of simplicity, this example only includes the arm with the highest reward for stage 2. Let us assume $\mathbb{P}(rw \| \mathcal{X}_0)$ and $\mathbb{P}(rf \| \mathcal{X}_0)$ return the highest reward at stage 1. We then define $\mathcal{A}_0^* = \{\mathbb{P}(rw \| \mathcal{X}_0), \mathbb{P}(rf \| \mathcal{X}_0)\}$, the arms for second stage, $\mathcal{A}_1$, are updated as follows:

$$\mathcal{A}_1 = \{\mathcal{A}_0^* \frown \mathbb{P}((rw) \| \mathcal{X}_1),\, ..., \mathcal{A}_0^* \frown \mathbb{P}((resub) \| \mathcal{X}_1)\,\} \tag{5}$$

where the sample from each arm in $\mathcal{A}_1$ will be a concatenation of two actions from $\mathcal{A}_0^*$ and $\mathbb{P}$. For stage 3, we simply replace $\mathcal{A}_0^*$ with $\mathcal{A}_1^*$, which will be the highest reward arms from $\mathcal{A}_1$. Finally, note that the subsets $\mathcal{X}_{0,1,2,3}$ are not necessary to be equally defined. For example, we can define $\mathcal{X}_0$= {`2×rw,b,rf,resub`}, $\mathcal{X}_{1,2}$ ={`rw, b, rf, resub`}, and $\mathcal{X}_3$={`b,rf,resub`}.

IV. FlowTune Framework

A. Initialization

The initialization step is crucial for MAB-based exploration performance. We leverage the domain knowledge of DAG-aware synthesis algorithms in our initialization stage. Specifically, for our multi-stage MAB exploration approach, we initialize the reward value for each arm in the first stage using the total number of transformable nodes by sampling each arm. While as demonstrated in Example 1, the total number of transformable nodes for a sequence of transformations highly depends on the first transformation, the initialization involves only one sampling of each arm. More importantly, this scheme significantly reduces the runtime of initialization for objectives such as technology mapping, since counting the number of transformable nodes does not require the actual mapping process. The initialization process is used at the beginning of each stage. To further improve the speed of initialization, we implement a parallel sampling function using OpenMP library [35].

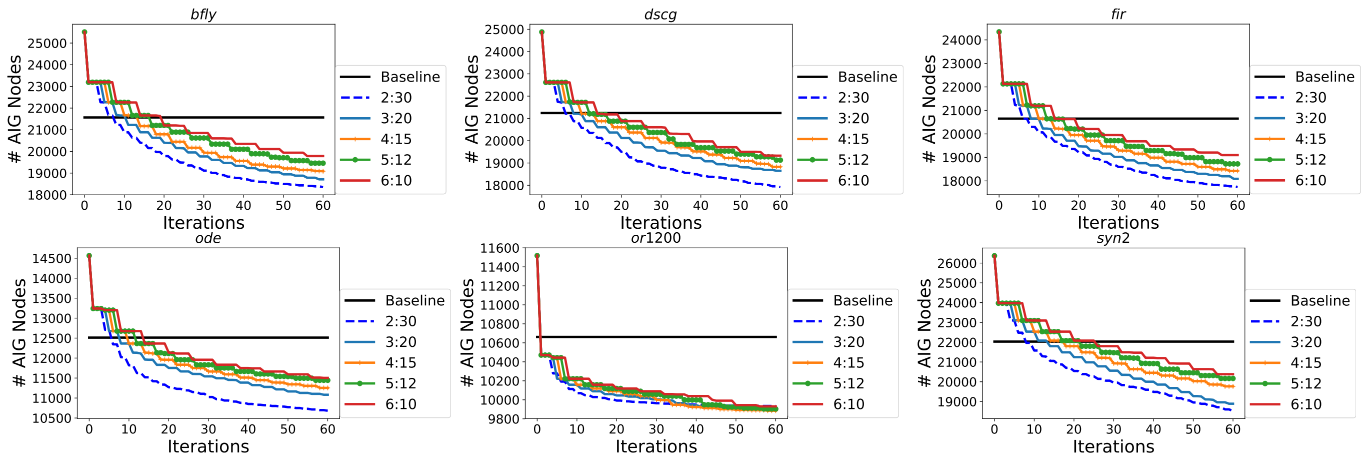


Fig. 5: *FTune* in AIG minimization (minimizing the number of AIG nodes). `Best-base` is the best baseline result obtained from hand-crafted scripts and ML-based approaches [6], [7]; s:m (e.g., 2:30) shows the # stages verses # iterations per stage.

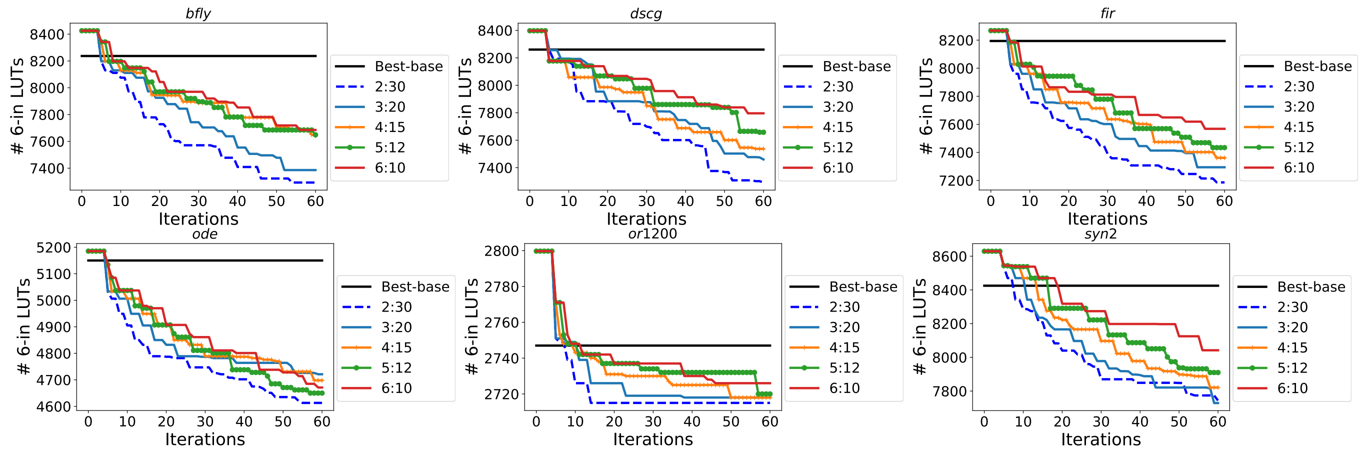


Fig. 6: *FTune* in LUT minimization for FPGA technology (6-in LUT devices). `Best-base` is the best baseline result obtained from hand-crafted scripts and ML-based approaches [6], [7]; s:m (e.g., 2:30) shows the # stages verses # iterations per stage.

B. System Integration

The proposed approach, namely *FlowTune*, is implemented in ABC and LSOracle [36]. Using the I/O interfaces of ABC and LSOracle, *FlowTune* can be applied to logic networks in various formats such as Verilog, AIG, and BLIF. The system overview is shown in Figure 4. Moreover, *FlowTune* has been integrated with ABC two technology mappers, i.e., `'map'` for standard-cell mapping, and `'if'` for FPGA mapping. Similarly, for MIG evaluation, *FlowTune* was integrated with LSOracle for optimization and native MIG mapping with `'lut_map'` command. For an accurate evaluation of *FlowTune*, timing and area results for standard-cell mapping, the mapped Verilog produced by ABC is evaluated using Cadence Genus with the predictive ASAP 7nm liberty [37]. Another considered point in our evaluation is on applying *FlowTune* for CNF clauses minimization. The size of CNF formulations is important to reduce the runtime of Boolean SAT solving, which has been widely applied in verification [38], [39], security [40], [41] and reasoning [42]. To this end, we use `write_cnf` function in ABC that dumps AIGs into CNF format. Finally, for FPGA evaluation, we apply *FlowTune* in two different contexts: (i) using the ABC interface built-in Yosys to generate mapped LUT-netlist, we evaluate *FlowTune* in Xilinx Vivado targeting Kintex UltraScale device; (ii) using OpenFPGA and VTR as back-end, we evaluate *FlowTune* targeting a Stratix IV like architecture. All the metrics are collected post-place and route, and we take as input for VTR optimized networks through ABC (for AIGs), and for OpenFPGA optimized networks with LSOracle (for MIGs).

V. Experimental Results

We demonstrate the proposed approach using six designs obtained from VTR 8.0 [3]. The experimental results are obtained using a CentOS 7 machine with a 48-core Intel Xeon operating at 2.1 GHz, 8 TB RAM, and 2 TB SSD. To show the flexibility and effectiveness of the proposed approach, we present results for different scenarios, from technology independent optimization to post mapping results for ASIC, and post-PnR results for FPGA. In particular, we present results for AIG minimization, standard cell technology mapping, post-PnR evaluation with Xilinx Vivado, post-PnR evaluation with VTR 8.0, and post-PnR assessment of MIGs + OpenFPGA flow. Table II presents the selected designs from VTR. It summarizes the number of I/O pins, the number of nodes, the logic depth, and a number of latches. In both cases, i.e., when we consider VTR flow with AIG, or MIG-based flow for OpenFPGA, the number of initial nodes are the same.

TABLE II: Details of selected VTR benchmarks for evaluating FlowTune. The designs are converted into BLIF format using VTR flow.

Design	I/O	Nodes	Latch	Level
bfly	482/257	28910	1748	97
dscg	418/257	28252	1618	92
fir	450/225	27704	1882	94
ode	275/169	16069	1316	98
or1200	588/509	12833	670	148
syn2	450/321	30003	1512	93

A. Evaluation of Logic Minimization

1) AIG Minimization: **Objectives for FlowTune**: **a)** minimizing the number of AIG nodes (Figure 5); and **b)** minimizing the number of 6-in LUTs for FPGA technology mapping (Figure 6). The benchmarks are listed in Table II. **Baselines**: Baseline results are collected using the hand-crafted scripted in ABC, and results produced using ML-based approaches proposed in [6], [7]. **Specifically, for the hand-crafted baseline, we applied hand-crafted flow `resyn` 25 times on all designs.** In Figures 5 and 6, `Best-base` is the best of these three baselines.

FlowTune setups: As proposed in Section III, we can configure FlowTune by changing the number of stages s, and the number of iteration m of each stage. Here, we evaluate five options of s:m, including `2:30, 3:20, 4:15, 5:12, 6:10`, such that the total number of iterations $s \cdot m$ holds identical.

AIG minimization results are shown in Figure 5 and LUT minimization minimization results are shown in Figure 6. We can see that, given the designs that ABC has larger design spaces, i.e., more design optimization potentials, FlowTune offers more optimizations over all the baselines. Another observation is that FlowTune outperforms baseline at early iterations for LUT minimization. That is because the DAG-aware synthesis algorithms are not developed to be LUT-mapping aware. Most of the algorithms implemented in ABC target minimizing the AIG nodes or the number of levels of AIG, which is known to have a miss-correlation with LUT-mapping results. More importantly, compared to the ML-based approaches [6], [7], for the same amount of exploration iterations, *FlowTune* is ~100 - 150× faster (runtime vary on design size), and also significantly outperforms these approaches in the QoR performance.

B. Standard Cell Technology Mapping

Objectives for FlowTune: Optimize the technology-mapping QoR, evaluated by Cadence Genus with gate sizing, using the ASAP 7nm library, while targeting **a)** STA delay optimization (Figure 7); and **b)** area optimization (Figure 8). QoR results are collected with Genus by importing the Verilog generated by `write_verilog` command in ABC. To the best of our knowledge, this is the first work that addresses synthesis flow tuning for STA-aware technology mapping. We can see that *FlowTune* effectively explores the design space by finding better synthesis flows for both area and post-STA delay optimization. However, *FlowTune* is not able to find any better synthesis flow after the initialization for design `or1200`. This is similar as shown in Figures 5 and 6. We believe the ABC synthesis flow design space of `or1200` is very limited for standard-cell technology mapping.

Unlike AIG and LUT mapping minimization, where FlowTune setup (s : m = 2:30) performs consistently better than others, we are not able to conclude a setup that likely will perform better than others for standard-cell mapping delay optimization. However, for standard-cell area minimization, we observe the same trend in results as for AIG and LUT minimization. While these observations offer some intuition for configuring *FlowTune* for different optimization objectives, we cannot provide formal explainability at this moment.

C. End-to-End Integration and Evaluation for FPGA Design

One of the greatest challenges of leveraging ML techniques in design flow optimization is to demonstrate the optimizations can be fully realized in an end-to-end design process, e.g., evaluating the QoR performance at the post-routing stage. Therefore, in this section, we evaluate *FlowTune* in two different end-to-end FPGA design frameworks, VTR 8.0 [43] and OpenFPGA [5], where our framework is integrated at the logic synthesis stage in both frameworks, for AIG and MIG. With this experiment, we aim to show the flexibility and portability of *FlowTune*, along with its capabilities to improve QoR. Thus, note that our goal is not to compare VTR and OpenFPGA, or AIGs and MIGs.

Evaluation metrics – To comprehensively evaluate the FPGA implementation performance, a list of QoR metrics that cover the area (utilization) and timing are selected. Specifically, our experimental results are conducted on measuring the post-routing (a) *total wirelength* (WL), (b) *total area* (Area) including logic area and routing area, (c) critical path delay, and (d) total negative slack (TNS). **Evaluation baselines** – The baseline results for VTR 8.0 are generated with the default settings collected from the VTR repository. Regarding the OpenFPGA framework, we integrate *FlowTune* in LSOracle, a state-of-the-art logic optimizer that handles MIG and AIGs. We focus on the MIG manipulation, and compare *FlowTune* against a high-effort MIG flow in LSOracle. OpenFPGA is used with its default settings. Both experiments target a Stratix IV-like FPGA architecture, which is common adopted modern architecture for FPGA works [44], [43]. In this context, each logic block is composed of 10 fracturable LUTs, and 200 routing tracks (channels).

Experimental setup for VTR 8.0 as backend – The experimental results conducted with VTR 8.0 as complete design flow involves the complete design flow from ODIN synthesis, with behavior Verilog HDL as inputs. The rest of the design flow includes logic synthesis and LUT-based technology mapping using ABC [1], in which *FlowTune* is plugged-in to optimize the design with automatically explored design flows. The output design will then be placed and

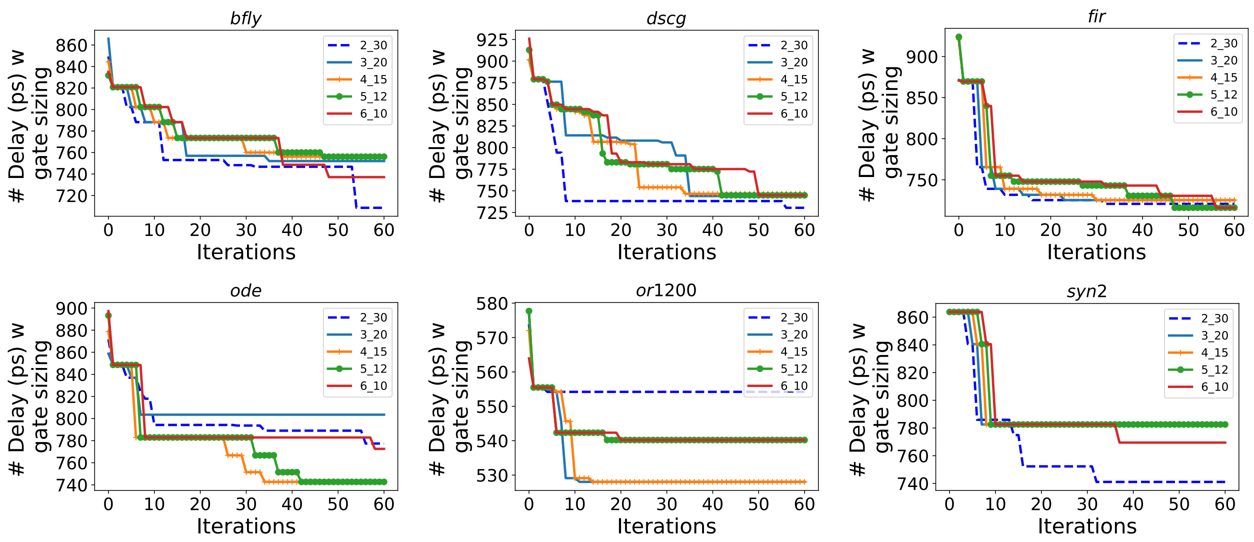


Fig. 7: *FTune* in STA delay optimization w gate sizing. QoR results are obtained using Genus with ASAP 7nm library.

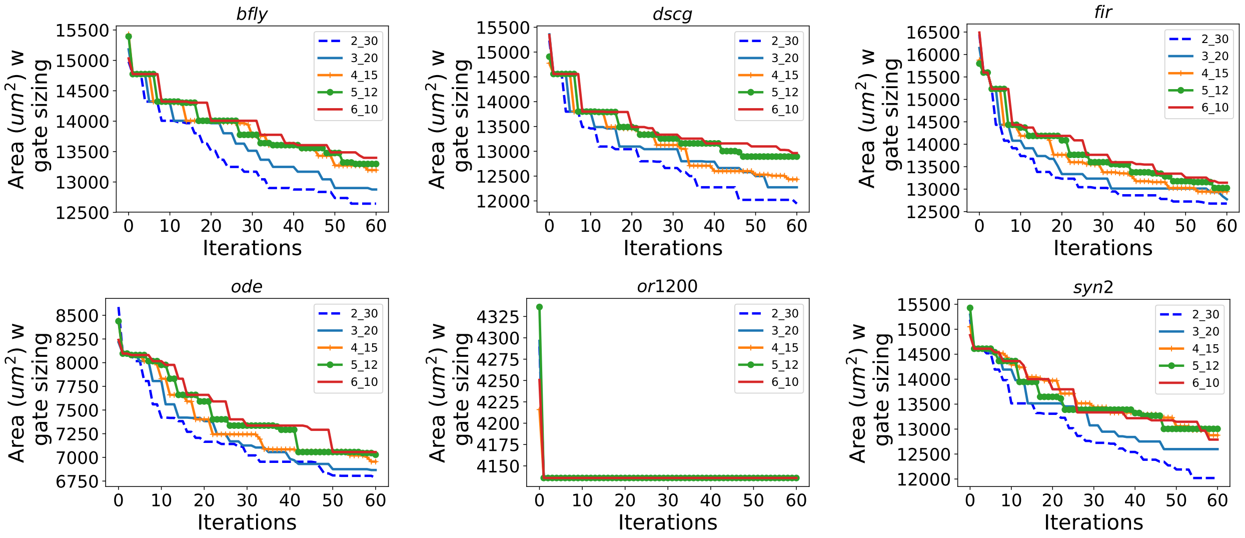


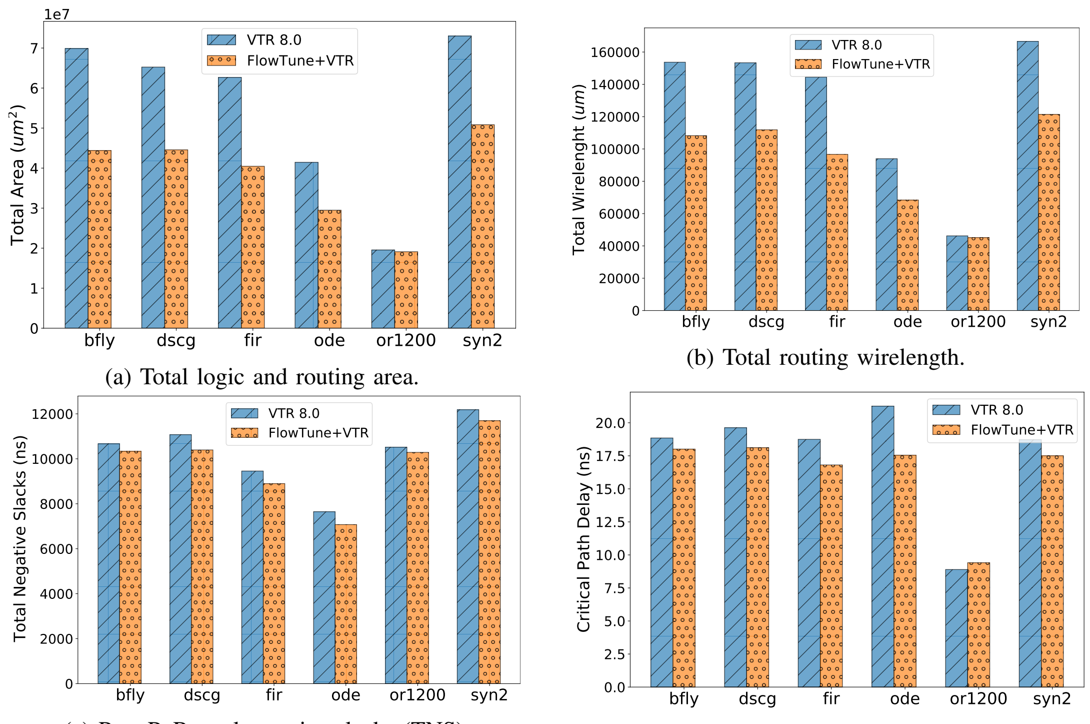
Fig. 8: *FTune* in area optimization using w gate sizing. QoR results are obtained using Genus with ASAP 7nm library.

routed w.r.t to the given FPGA architecture using VPR default settings. The results included in this section are all collected at the post-routing stage.

Experimental setup for OpenFPGA as backend – The OpenFPGA design flow also includes the complete design flow from ODIN synthesis, with behavior Verilog HDL as inputs. Then it uses LSOracle to read an input BLIF into a MIG, and perform logic optimization. *FlowTune* is integrated with LSOracle to autonomously generate a MIG-based flow. LSOracle is then used for LUT mapping, and the output is dumped into a BLIF file. The BLIF is then used as input to OpenFPGA with default settings, and all the results are collected post-routing. **Note that LSOracle synthesis framework is MIG-based logic synthesis engine, where the transformations are all based on MIG. These experiments aim to demonstrate the DAG-based synthesis domain-specific knowledge extracted from AIG optimization is transferable to MIG as well.**

To show the flexibility of our approach, we present results for two different end-to-end FPGA design evaluations, i.e., FlowTune in VTR 8.0 and FlowTune in LSOracle + OpenFPGA. Note that we do not intend to compare the differences between logic synthesis data structures (i.e., AIG and MIG), and do not intend to compare the performance across different backend frameworks (VTR and OpenFPGA). In summary, our goal with these experiments is to show that FlowTune can be easily integrated and verify its effectiveness in different scenarios.

Results on total area – (1) **VTR results** – Figure 9a shows the results of post-routing evaluation on design area which is the sum of total used logic block area and the total routing area. When designs are evaluated with *FlowTune*, the area can be reduced by $\sim 30\%$ on average for `bfly`, `dscg`, `fir`, `ode`, and `syn2`. However, for design `or1200`, we can achieve few improvement results with *FlowTune* optimization. (2) **OpenFPGA results** – OpenFPGA results show area improvements in all the cases, as seen in Figure 10a. Results present up to 10.0% area improvement, with an average of 4.5% area



(a) Total logic and routing area.

(b) Total routing wirelength.

(c) Post-PnR total negative slacks (TNS).

(d) Post-PnR critical path delay.

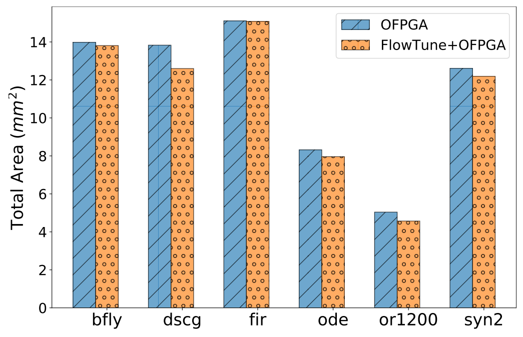
Fig. 9: Post-routing evaluation with VTR (VPR PnR) as backend using six benchmarks collected from [3], with default VTR 8.0 flow as baseline where logic synthesis is conducted on **AIG logic optimization**. The collected results include (a) total area including logic and routing area, (b) total routing wire length, (c) post-PnR total negative slacks (TNS), and (d) post-PnR critical path delay.

reduction. While these results are not as expressive as in the VTR flow, it is still relevant and consistent among both flows.

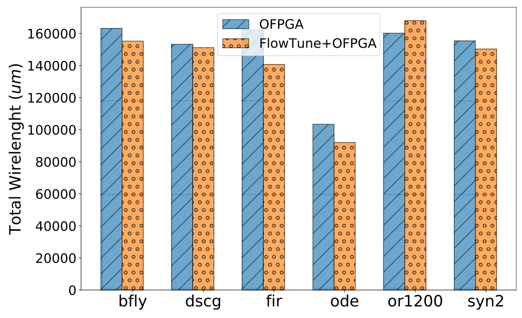
Results on total wirelength – (1) **VTR results** – We further analyze the total wirelength of the post-PnR designs to clearly show that routing has been improved, in addition to the logic optimization results. Figure 9b shows the results of post-routing evaluation on total routing wirelength. The performance with *FlowTune* can be improved for all designs. Similarly, for total area evaluation, the improvement is marginal for design `or1200`. (2) **OpenFPGA results** – Figure 10b presents the total wirelength for MIGs with OpenFPGA. In this case, *FlowTune* presents better results in 4 cases, with little overhead in two designs (up to 4%). On the other hand, *FlowTune* reduces the total wirelength in up to 14%, and 5% on average. As the goal of *FlowTune* is to reduce the logic area, it might reflect in longer total wirelength in some cases due to the improved logic sharing. Still, the overhead compared to the baseline flow was small for the considered designs. We can observe that *FlowTune* was able to consistently reduce total wirelength in both flows for almost all the benchmarks. When considering `or1200`, in the VTR flow, *FlowTune* had minor gains compared to the baseline, whereas in OpenFPGA it had some overhead compared to the baseline.

Results on timing – Post-PnR timing results are presented with critcal path delay and total negative slacks (TNS), which are the most critical two metrics used for timing evaluation. (1) **VTR results** – The performance with *FlowTune* can be improved for all designs except for `or1200`. On the other hand, we have observed that the `or1200` timing performance has a slight improvement for TNS but got worse than the default in critical path delay. While there has been very little logic reduction (see Figure 6) and wirelength reductions, the structure of the `or1200` design remains almost the same. We believe that the critical path delay and TNS results differences between w and w/o FlowTune is from the randomness of the placement and routing algorithms in VPR. (2) **OpenFPGA results** – Figure 10d presents the worst negative slack for the considered benchmarks. We can see that *FlowTune* greatly improves the baseline. In particular, we improve all the cases, with up to 37% gains for the *or1200*. Still, five designs have over 7% gains in the WNS, showing the effectiveness of *FlowTune* in improving MIGs over a state-of-the-art recipe. On average, *FlowTune* reduces the delay by 7% on average. Therefore, *FlowTune* achieves great delay improvements in both flows. The main difference is that while in VTR flow it could not benefit the `or1200`, in the OpenFPGA flow the `or1200` was greatly improved. Figure 10c presents the OpenFPGA total negative slack results. We improved the baseline in 5 cases, up to 12%. For one case, we had a 1% TNS overhead. On average, we presented an average TNS reduction of 5%. Total negative slack has a similar trend in both, with gains when applying *FlowTune*.

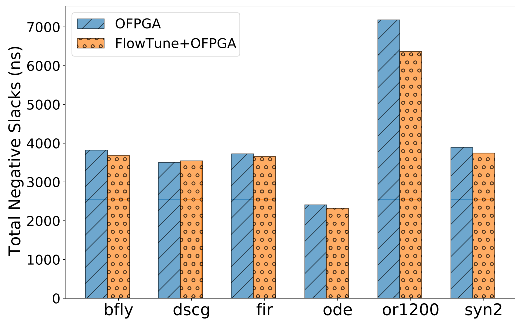
In conclusion, we can see that FlowTune framework with the domain-specific MAB algorithm can flexibly and effec-



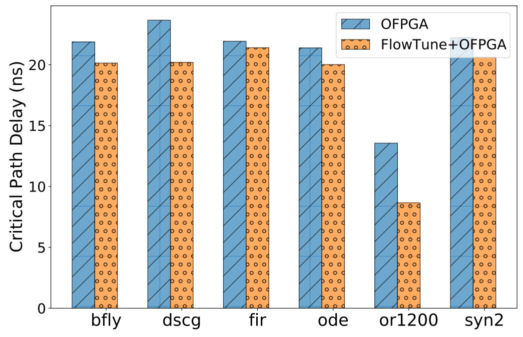
(a) Total logic and routing area.



(b) Total routing wirelength.



(c) Post-PnR total negative slacks (TNS).



(d) Post-PnR critical path delay.

Fig. 10: Post-routing evaluation with LSOracle+OpenFPGA where the logic synthesis process involves **MIG logic optimization**, with OpenFPGA as backend using the same benchmarks collected from [3]. The results are compared to default LSOracle+OpenFPGA flow, including (a) total area including logic and routing area, (b) total routing wire length, (c) post-PnR total negative slacks (TNS), and (d) post-PnR critical path delay.

tively navigate flow optimizations in different logic synthesis DAG representations, and different PnR backends, and yet produce effective and consistent results. That positions *FlowTune* as a versatile, light-weight, and portable flow exploration framework.

VI. Related Work

Recently, we have seen significant progress in leveraging ML techniques for logic synthesis. Specifically for exploring synthesis flows as sequential decision making problem, there are mainly two directions – (1) learning a static ML-based predictor to enable fast design space and (2) exploring flows in reinforcement and iterative fashion.

In [6], Yu et al. proposed the first ML-based flow exploration approach, which involves a CNN-based QoR predictor to enable fast flow exploration and generation. They model the problem as a multi-class classification problem, and the CNN outputs angel- and devil-flows, where angel flows produce the best QoR results and devil flows likely offers the worst QoR. While this approach learns a static ML model that eliminates the expensive runtime of evaluating a large number of flows in synthesis tool, one critical drawback is on the data collecting and labelling. Although, the authors proposed a follow-up approach based on recurrent neural networks and transfer learning to reduce the efforts in collecting labelled dataset, this approach is limited on limited technology domain, which is limited on generalizability for technology transferability [24].

To walk around the challenge of labelled data collection, reinforcement learning (RL) approaches has then been adopted for logic synthesis flow generation. Liu et. al. [45] first cast logic optimization as a *Markov Decision Process* (MDP), where a set of logic transformations could be chosen in the next iteration of the synthesis flow. However, it has been demonstrated that the performance of a sequence of logic synthesis transformations does not satisfy MDP properties, since the performance of each transformation does not solely depend on previous transformation. Hosny et al. [7] propose a Deep RL-based approach that aims to optimize the area given a timing constraint. They cast the generation of logic synthesis flow into a game-like problem, where the actions are the set of transformations to be selected. However, the RL training process is very time consuming and offers poor flexibility (e.g., limited flow lenghth, QoR-specific RL model, etc.,) and integration capability.

To further ehance the structure information in the ML-based approaches for flow exploration, there have recently seen many works leveraging Graph Neural Network (GNNs) to improve the generalizability. Zhu et. al [46] propose to model the logic synthesis flow generation as MDP problem and use GNNs to enhance the state representation. Therefore, besides AIG statistics and the history of transforms applied, they also aggregate the graph-structure through GNNs. They present improvements over the ABC *resyn2* flow, with the same length of transformations to be applied. Similarly, in [47], the authors combine RL and GNN to optimize MIGs. In addition to utilize the feature extraction capability of GNNs, Wu [9] et. al. demonstrates that GNNs can be used to aggregate structure features such that static ML approaches can be trained with significantly reduced labeled data. Unfortunately, these work are only evaluated at the stage of logic-level without considering the impacts of low-level design stages such as placement and routing.

VII. Conclusion

This paper presents a generic end-to-end and high-performance domain-specific, multi-stage multi-armed bandit framework for Boolean logic optimization. Specifically, we propose a first-of-its-kind synthesis flow exploration algorithm based on MAB that utilizes domain-specific knowledge of DAG-aware synthesis algorithms. A comprehensive analysis of the DAG-aware algorithms in synthesis flows is provided to demonstrate the importance of the extracted domain knowledge. Moreover, we propose a novel MAB mechanism, including a fast initialization and multi-stage MAB exploration approach. To demonstrate the performance and the flexibility of our framework, we build a complete exploration framework that integrates with several tools. Our results on AIG minimization, LUT minimization, CNF minimization, post static timing analysis (STA) delay and area optimization for standard-cell technology mapping, and end-to-end PnR evaluation with different backend tools, have demonstrated that FlowTune outperforms the existing works in both optimization performance and runtime. We also demonstrate that our domain-specific MAB algorithm is generalizable to different DAG-based logic synthesis techniques, where we have integrated FlowTune for both AIG and MIG optimizations. Future work will focus on explainability and robustness analysis of ML-based design space exploration, and architecture-aware optimizations.

Acknowledgement This work is funded by National Science Foundation (NSF) awards NSF-2008144 and NSF-2047176.

References

[1] A. Mishchenko *et al.*, "ABC: A System for Sequential Synthesis and Verification," *URL http://www. eecs. berkeley. edu/alanmi/abc*.

[2] C. Wolf, "Yosys Open Synthesis Suite," 2016.

[3] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "VTR 8.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[4] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "Lsoracle: A logic synthesis framework driven by artificial intelligence," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–6.

[5] X. Tang, E. Giacomin, A. Alacchi, B. Chauviere, and P.-E. Gaillardon, "Openfpga: An opensource framework enabling rapid prototyping of customizable fpgas," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 367–374.

[6] C. Yu, H. Xiao, and G. D. Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, 2018, pp. 50:1–50:6.

[7] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "Drills: Deep reinforcement learning for logic synthesis," *arXiv preprint arXiv:1911.04021*, 2019.

[8] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "LAMDA: Learning-Assisted Multi-stage Autotuning for FPGA Design Closure," in *FCCM'19*.

[9] N. Wu, J. Lee, Y. Xie, and C. Hao, "Hybrid graph models for logic optimization via spatio-temporal information," *arXiv preprint arXiv:2201.08455*, 2022.

[10] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 129–132.

[11] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for fpga hls using graph neural networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[12] W. L. Neto, M. T. Moreira, Y. Li, L. Amarù, C. Yu, and P.-E. Gaillardon, "Slap: A supervised learning approach for priority cuts technology mapping," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 859–864.

[13] W. L. Neto, M. T. Moreira, L. Amaru, C. Yu, and P.-E. Gaillardon, "Read your circuit: Leveraging word embedding to guide logic optimization," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021, pp. 530–535.

[14] C. Yu and Z. Zhang, "Painting on placement: Forecasting routing congestion using conditional generative adversarial nets," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[15] C. Yu, "Flowtune: Practical multi-armed bandits in boolean optimization," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[16] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2020, pp. 145–150.

[17] Y. V. Peruvemba, S. Rai, K. Ahuja, and A. Kumar, "Rl-guided runtime-constrained heuristic exploration for logic synthesis," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[18] N. Kapre, H. Ng, K. Teo, and J. Naude, "InTime: A Machine Learning Approach for Efficient Selection of FPGA CAD Tool Parameters," *FPGA*, Feb. 2015.

[19] H. Liu and L. P. Carloni, "On Learning-based Methods for Design-space Exploration with High-level Synthesis," *DAC*, Jun. 2013.

[20] G. Pasandi, S. Nazarian, and M. Pedram, "Approximate logic synthesis: A reinforcement learning-based technology mapping approach," in *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019, pp. 26–32.

[21] M. M. Ziegler, R. B. Monfort, A. Buyuktosunoglu, and P. Bose, "Machine Learning Techniques for Taming the Complexity of Modern Hardware Design," *IBM Journal of Research and Development*, vol. 61, no. 4, p. 13, 2017.

[22] D. Li, S. Yao, Y. Liu, S. Wang, and X. Sun, "Efficient Design Space Exploration via Statistical Sampling and AdaBoost Learning," *DAC*, Jun. 2016.

[23] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutional networks with applications in testability analysis," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[24] C. Yu and W. Zhou, "Decision making in synthesis cross technologies using lstms and transfer learning," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 55–60.

[25] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[26] S. Liu, F. C. Lau, and B. C. Schafer, "Accelerating fpga prototyping through predictive model-based hls design space exploration," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[27] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen *et al.*, "Routenet: routability prediction for mixed-size designs using convolutional neural network," in *ICCAD*. ACM, 2018, p. 80.

[28] B. Xu, Y. Lin, X. Tang, S. Li, L. Shen, N. Sun, and D. Z. Pan, "WellGAN: Generative-Adversarial-Network-Guided Well Generation for Analog/Mixed-Signal Circuit Layout," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 66.

[29] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Dag-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, 2006, pp. 532–535.

[30] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization,"

in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[31] "OpenCores," *URL https://opencores.org*.

[32] C. Yu, M. J. Ciesielski, M. Choudhury, and A. Sullivan, "Dag-aware logic synthesis of datapaths," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 2016, pp. 135:1–135:6.

[33] L. Stok, D. Kung, and et al., "BooleDozer: Logic Synthesis for ASICs," *IBM Journal of Research and Development*, vol. 40, no. 4, pp. 407–430, 1996.

[34] C. Yu, M. Choudhury, A. Sullivan, and M. J. Ciesielski, "Advanced datapath synthesis using graph isomorphism," in *ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, 2017, pp. 424–429.

[35] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[36] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "Lsoracle: a logic synthesis framework driven by artificial intelligence: Invited paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–6.

[37] X. Xu, N. Shah, A. Evans, S. Sinha, B. Cline, and G. Yeric, "Standard cell library design and optimization methodology for asap7 pdk," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 999–1004.

[38] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.

[39] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[40] C. Yu, X. Zhang, D. Liu, M. Ciesielski, and D. Holcomb, "Incremental sat-based reverse engineering of camouflaged logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 10, pp. 1647–1659, 2017.

[41] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 137–143.

[42] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[43] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. Kent, and J. Rose, "Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, p. 32, 2011.

[44] G. Gore, X. Tang, and P.-E. Gaillardon, "A scalable and robust hierarchical floorplanning to enable 24-hour prototyping for 100k-lut fpgas," in *Proceedings of the 2021 International Symposium on Physical Design*, 2021, pp. 135–142.

[45] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for lut-based technology mapping," *FPGA'17*, 2017.

[46] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 145–150.

[47] X. Timoneda and L. Cavigelli, "Late breaking results: Reinforcement learning for scalable logic optimization with graph neural networks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1378–1379.