FlowTune: Practical Multi-armed Bandits in Boolean Optimization

Cunxi Yu
University of Utah
cunxi.yu@utah.edu

ABSTRACT

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 for training or are limited in practical EDA toolflow integration due to computation overhead. This paper presents a generic end-to-end and high-performance domain-specific, multi-stage multi-armed bandit framework for Boolean logic optimization. This framework addresses optimization problems on **a)** And-Inv-Graphs (# nodes), **b)** Conjunction Normal Form (CNF) minimization (# clauses) for Boolean Satisfiability, **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, the proposed framework has been integrated with ABC [1], Yosys [2], VTR [3], and industrial tools. The experimental results demonstrate that our framework outperforms both hand-crafted flows [1] and ML explored flows [4, 5] in quality of results, and is orders of magnitude faster compared to ML-based approaches [4, 5].

1 INTRODUCTION

Targeted specialization of functionality in hardware has become arguably the best means for enabling improved compute performance and energy efficiency. However, as the complexity of modern hardware systems explodes, fast and effective hardware explorations are hard to achieve due to the lack of guarantee in the existing in electronic design automation (EDA) toolflow. Several major limitations prevent practical hardware explorations [6–8]. First, as the hardware design and technology advance, the design space of modern EDA tools has increased dramatically. Besides, evaluating a given design point is extremely time-consuming, such that only a very small sub-space of the large design space can be explored. Last but not least, while initialization of design space exploration is important for the final convergence, it is difficult to initialize the search for unseen designs effectively.

Recent years have seen increasing employment of decision intelligence in EDA, which aims to reduce the manual efforts and boost the design closure process in modern toolflows [4–7, 9–13]. For example, various of machine learning (ML) techniques have been used to automatically configure the tool configurations of industrial FPGA toolflow [6, 9, 10, 14, 15] and ASIC toolflow [4, 5, 7, 12]. These works focus on end-to-end tool parameter space exploration, which are guided by ML models trained based on either offline [7] or online datasets [6, 9]. Moreover, exploring the sequence of synthesis transformations (also called synthesis flow) in EDA has been studied in an iterative training-exploration fashion through Convolutional Neural Networks (CNNs) [4] and reinforcement learning [5]. While the design quality is very sensitive to the sequence of transformations [4], these approaches are able to learn a sequential decision making strategy to achieve better quality of results [4, 5]. In addition, neural network based image classification and image construction techniques have been leveraged in placement and route (PnR), in order to accelerate design closure in the physical design stage [16–22]. While ML-based approaches have shown promising results in different EDA stages, such systems have limited practical applications. The major limitations include:

- **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 [4, 5, 9, 10].
- **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) [4, 5, 9].
- **System integration overhead**. There is significant runtime overhead while EDA tools communicate with machine learning frameworks (e.g., TensorFlow used in [4, 5, 16, 18]).

To overcome these limitations, we propose *FlowTune* (*FTune*), a generic end-to-end, high-performance, and practical domain-specific multi-armed bandit (MAB) approach for Boolean logic optimization. *FTune* is implemented in the synthesis and verification framework ABC [1]. Using the interfaces in ABC, *FTune* has been integrated with several toolflows, including VTR 8.0 [3], Yosys [2], Cadence Genus, and Xilinx Vivado. To demonstrate the performance, scalability and flexibility, we have applied *FTune* to various Boolean logic optimization problems, targeting *And-Inverter-Graphs* (AIG) [23], STA-timing aware standard-cell (STD) technology mapping, FPGA technology mapping, and *Conjunction Normal Form* (CNF) for Boolean Satisfiability (SAT) [24]. The main contributions of this work include:

ICCAD '20, November 2–5, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415615

• A ovel domain-specific bandit algorithm for sequential decision-making by leveraging domain knowledge of DAG-aware synthesis algorithms (Section 3).
• *FTune* is evaluated with several integrated circuit (IC) optimization problems for both FPGAs and ASICs (Section 4) and outperform ML-based approaches and hand-crafted heuristics.
• Moreover, this is the first work that addresses *static timing analysis* (STA) aware technology mapping (using 7nm ASAP [25] library) and CNF minimization.
• The benefits to the industrial flow are demonstrated by applying *FTune* in Vivado flow, where the QoRs are evaluated after PnR.
• **FlowTune framework and benchmarks have been fully released publicly** [1].

2 BACKGROUND

2.1 Boolean Logic Optimization

A Boolean circuit can be represented using a directed acyclic graph (DAG) with nodes representing logic gates and directed edges representing wires connecting the gates. The most efficient algorithms that optimize the Boolean circuit are based DAG-aware Boolean transformations [1, 2, 23, 26–29], which are implemented based on And-Inverter-Graphs (AIGs). AIGs are DAGs where each node represents logic AND function, and the weighted edges indicate the inversions of the Boolean signals. For example, the synthesis transformations used in the existing ML-based exploration work [4, 5] are all DAG-aware synthesis algorithms implemented on AIGs. A common algorithmic structure of DAG-aware synthesis algorithms includes two main parts: 1) *graph traversing* – the DAG-aware synthesis algorithms first traverse the DAG and search for the nodes can be optimized by applying certain transformations; 2) *apply transformations* – algorithms then apply those transformations on the transformable nodes and update the DAG (pseudo-code in Figure 1). A large number of DAG-aware synthesis algorithms are developed for a variety of optimization objectives.

2.2 Synthesis Flows and Exploration

Synthesis flows are a set of synthesis transformations that apply iteratively to the Boolean network or circuit design, which are mostly a combination of DAG-aware synthesis algorithms. The synthesis transformations are mainly involved in three stages of the design flow: high-level synthesis (HLS), logic synthesis (LS) and placement and route (PnR). For different types of electronic designs, it is well demonstrated that synthesis flows need to be carefully customized for optimal performance[4, 5, 30]. In this work, we focus on logic synthesis flows exploration.

Exploration of synthesis flows has been considered as a sequential decision-making problem, as synthesis flows involve a sequence of synthesis transformations. Recently, deep learning techniques have shown promising synthesis flow exploration results compared to hand-crafted synthesis flows [4, 5]. However, due to the large search space, expensive labeled data collection, and high system integration overhead, those approaches are not practical. Moreover, previous works consider flows with all synthesis transformations applied same times, i.e., namely *m-repetition* flows where each transformation is applied m times. Thus, Yu et. al [4] limits the theoretical search space of flow exploration to a *multi-set* permutation problem. For m-repetition flows with n unique transformations, the exploration space is $\frac{(n \cdot m)!}{(m!)^n}$. In this work, we consider a more general flow space that covers arbitrary types of flows. Specifically, let n be the number of synthesis transformations, the M-repetition flows, M=$\{m_1, m_2, ..., m_n\}$, where m_i is the number of repetitions of the i^{th} ($i <= n$) transformation. The theoretical exploration space is shown in Equation 1.

$$S(M,n) = \frac{(m_1 + m_2 + \cdots m_n)!}{(m_1!)(m_2!)\cdots(m_n!)} \quad (1)$$

2.3 Technology Mapping

Technology mapping is the problem of implementing a logic circuit using a set of specific components of a technology library. Mostly, combinational logic components and memory components are used to implement the sequential circuits. A standard-cell library would typically consist of gates of varying gate sizes for primitive logic functions, where the area, power, and delay can be very different for the same functionality. For FPGA technology mapping, the logic circuit will be mapped into a K-LUT (LookUp-Table) network where K is fixed by the target FPGA device using a LUT library. The challenge is to construct a mapping that maximally utilizes the gates in the library to implement the logic function of the circuit and achieve some performance goal, e.g., minimizing critical path delay. Note that most of the technology mapping algorithms focus on optimizing the delay and area using static technology models, and do not consider the sizes of the gates (i.e., gate sizing). However, the QoRs after gate sizing with statistical timing analysis (STA) can be dramatically changed. Hence, in this work, for ASIC flow, we evaluate our framework using post-STA QoR with gate sizing; for FPGA flow, we focus on minimizing the number of LUTs.

2.4 Boolean SAT and CNF

In Boolean logic, a formula is in conjunctive normal form (CNF), which is a conjunction of one or more clauses, where a clause is a disjunction of literals. In other words, CNF is formed with a logic AND of many logic ORs. All conjunctions of literals and all disjunctions of literals are in CNF, as they can be seen as conjunctions of one-literal clauses and conjunctions of a single clause, respectively. Boolean SAT problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula, where the formula is in CNF format. SAT is the first problem that was proven to be NP-complete. Current state-of-the-art SAT solvers use conflict-driven clause learning (CDCL) algorithm for solving the SAT problems, where the number of clauses in the CNF is an important factor in the performance. While CNF is a conjunction (AND) of disjunctions (OR) of literals (Boolean signals), the number of clauses can be minimized using logic synthesis flows as well. Hence, in this work, we leverage our framework into CNF minimization for model checking.

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

```
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
```

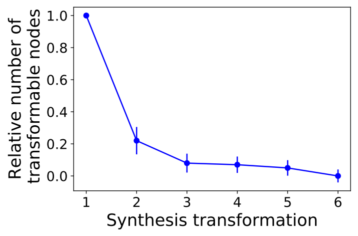

Figure 1: Illustration of DAG-aware synthesis algorithm. And, the relative number of AIG nodes that are effectively executed in each transformation of the synthesis flows, using 100 random flows with six transformations used in [4, 5].

3 APPROACH

3.1 Algorithmic Domain Knowledge of DAG-Aware Synthesis

The most efficient algorithms that optimize the Boolean networks are directed acyclic graph (DAG) aware based Boolean synthesis algorithms [2, 23], which are widely used in both open-source tools [2**?** , 3] and industrial tools [26, 27, 31, 32]. 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 1) 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 as [4, 5].

The analysis results are shown in Figure 1, 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 flow with six transformations, 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 1, the relative number of transformed nodes of all randoms flows start with 1. While there are 100 random permuted flows are used in this analysis, the errorbars are used to indicate the upper/lower bound.

There are two main observations from this analysis:

- 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 1, 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.

Table 1: Example of the algorithmic domain knowledge presented in Figure 1 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**

- 2) The optimization performance of synthesis flow is dominated by the first transformation since the first transformation 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 bad QoR regardless of the permutation of the rest flow. 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. These two observations are the main motivations leading to the proposed domain-knowledge MAB approach.

Example 1: An illustrative example that demonstrates the importance of the extracted algorithmic domain knowledge presented in Figure 1 is shown in Table 1. 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`.

3.2 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 limited set of resources must be allocated between available choices in a way that maximizes their expected gain of a given objective.

These problems are the principal theoretical tool for modeling the exploration-exploitation tradeoffs inherent in 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 $\mathcal{A}$={$resyn, rewrite, ..., refactor$}. 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 is playing with each transformation (each arm) with enough rounds so as to eventually get the true probability of reward, which is not practical. 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. Although the classic MAB sequential decision making has been widely applied in many applications, it has several significant drawbacks in logic synthesis flow exploration according to the two observations in Section 2.1:

- 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 since the logic graph will be updated by the transformation.

In order to gathering the domain knowledge of synthesis algorithms discussed in Section 2.1, we propose a novel MAB environment by re-defining the arms and actions. Thus, let $\mathbb{P}(\mathcal{X})$ 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 that $\mathbb{P}(x\|\mathcal{X})$ 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 set $\mathcal{X}$. Let $Q(a_t)$ be the action value that is obtained by applying a_t to the given logic circuit 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 upper bound in this work is shown below.

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

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 will be achieved

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

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

3.3 Multi-stage Bandit

While the approach described in the previous section focuses on optimistic initialization, we propose a multi-stage 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, while increasing the length of the sequences, a more significant number of explorations are required to close the gap between the optimistic reward distribution $\mathcal{R}^*$ and exploitation reward distribution $\mathcal{R}$. Moreover, the optimistic reward distribution $\mathcal{R}^*$ will change once the synthesis transformations are applied to the logic circuit since the graph structure has been changed. Finally, although the synthesis transformations are less effective at late time steps, a fine-grain exploration can

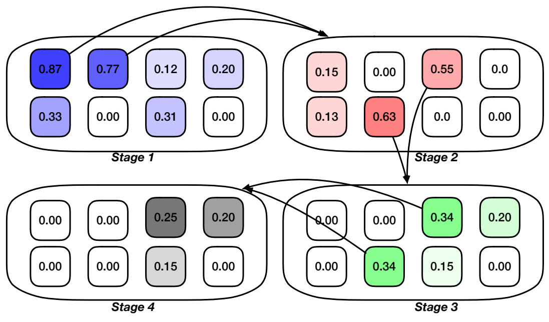

Figure 2: Illustration of the proposed multi-stage bandit approach with four stages.

potentially be very impactful to later stages in the EDA flows, such as technology mapping and gate sizing.

We present our multi-stage bandit approach using a four-stage example shown in Figure 2. Each stage in the multi-stage approach uses the same domain-specific bandit algorithm described in Section 3.2. Within each stage, the MAB algorithm is restricted to a fixed number of iterations m. Once the first stage is completed, exploitation reward distribution $\mathcal{R}^1$ will be updated (stage 1 in Figure 2), where a higher rate indicates the higher chance 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 2, $\mathcal{R}^1_{a_0}$ 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 aforementioned domain-specific MAB and the muilti-stage bandit algorithm for exploring synthesis flows using 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 $\mathcal{A}_0$

$$\mathcal{A}_0 = \{\, \mathbb{P}(rw \| \mathcal{X}_0), \mathbb{P}(rf \| \mathcal{X}_0), ..., \mathbb{P}(resub \| \mathcal{X}_0)\,\} \tag{2}$$

will be explored with our MAB algorithm presented in Section 3.2. The reward r_t is calculated based on the targeting objective, e.g., number of AIG nodes newly reduced compared to existing best actions. Stage 1 will terminate after a fixed number of iterations, and returns the arm(s) with highest reward value. In this example, we only include the arm with highest reward for stage 2. Assume $\mathbb{P}(rw \| \mathcal{X}_0)$ and $\mathbb{P}(rf \| \mathcal{X}_0)$ returns the highest reward at stage 1, we define $\mathcal{A}^*_0 = \{\mathbb{P}(rw \| \mathcal{X}_0), \mathbb{P}(rf \| \mathcal{X}_0)\}$, the arms for second stage, $\mathcal{A}_1$, will be updated as follows:

$$\mathcal{A}_1 = \{\mathcal{A}^*_0 \frown \mathbb{P}((rw) \| \mathcal{X}_1), ..., \mathcal{A}^*_0 \frown \mathbb{P}((resub) \| \mathcal{X}_1)\,\} \tag{3}$$

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 $\mathcal{A}_1$. 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`}.

3.4 Initialization

The initialization (warm-up) 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.

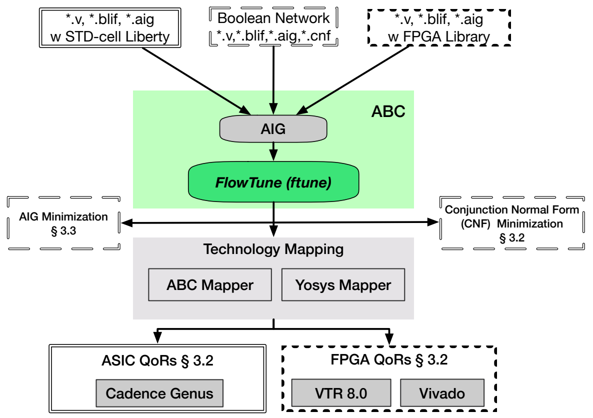
*.v, *.blif, *.aig w STD-cell Liberty
Boolean Network *.v,*.blif,*.aig,*.cnf
*.v, *.blif, *.aig w FPGA Library
ABC
AIG
FlowTune (ftune)
AIG Minimization § 3.3
Conjunction Normal Form (CNF) Minimization § 3.2
Technology Mapping
ABC Mapper
Yosys Mapper
ASIC QoRs § 3.2
Cadence Genus
FPGA QoRs § 3.2
VTR 8.0
Vivado

Figure 3: 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.

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 [33].

3.5 System Integration

The proposed approach, namely *FTune*, is implemented in ABC. Using the I/O interfaces of ABC, *FTune* can be applied to logic networks in various formats such as Verilog, AIG, and BLIF. The system overview is shown in Figure 3. Moreover, FTune has been integrated with ABC two technology mappers, i.e., `'map'` for STD mapping, and `'if'` for FPGA mapping. For evaluating *FTune*, the timing and area results of optimizing STD mapping, the mapped Verilog produced by ABC are evaluated using Cadence Genus with ASAP 7nm liberty. The size of CNF formulations is important for the runtime performance of Boolean SAT solving, which has been widely applied in verification [24, 34], security [35, 36] and reasoning [37]. Regarding *FTune* for CNF clauses minimization, we use `write_cnf` function in ABC that dumps AIGs into CNF format. Finally, using the ABC interface built-in Yosys to generate mapped LUT-netlist, we evaluate *FTune* in Xilinx Vivado targeting Kintex UltraScale device.

4 RESULT

We demonstrate the proposed approach using eight DSP 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. The results are collected with various Boolean logic optimization objectives, including logic minimization (Section 4.1), STD technology mapping optimization

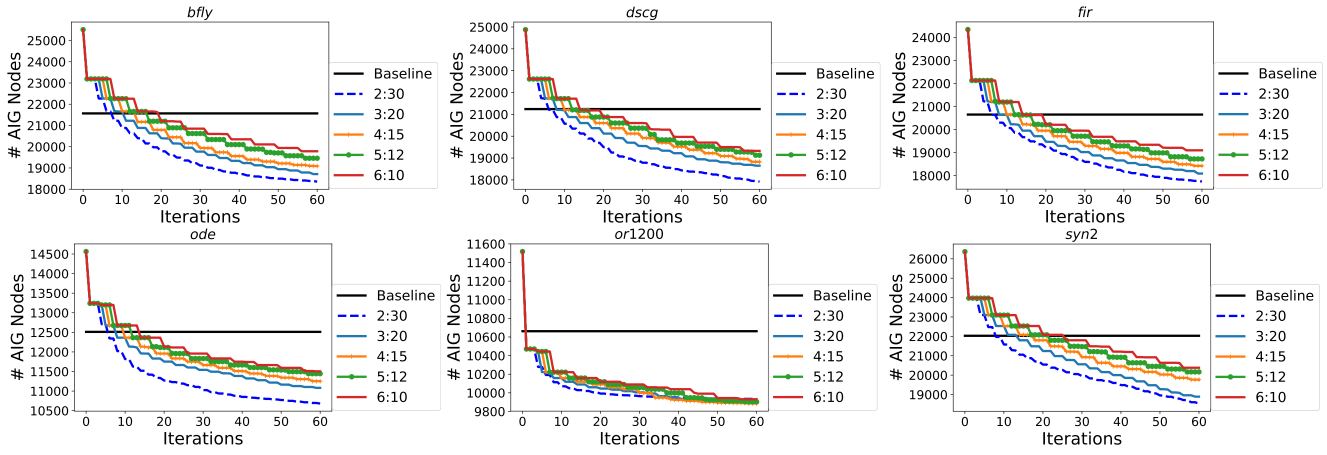

Figure 4: *FTune* in AIG minimization (minimizing the number of AIG nodes). `Best-base` is the best baseline result obtained from hand-crafted scripts and ML-based approaches [4, 5]; *s*:*m* (e.g., 2:30) shows the # stages verses # iterations per stage.

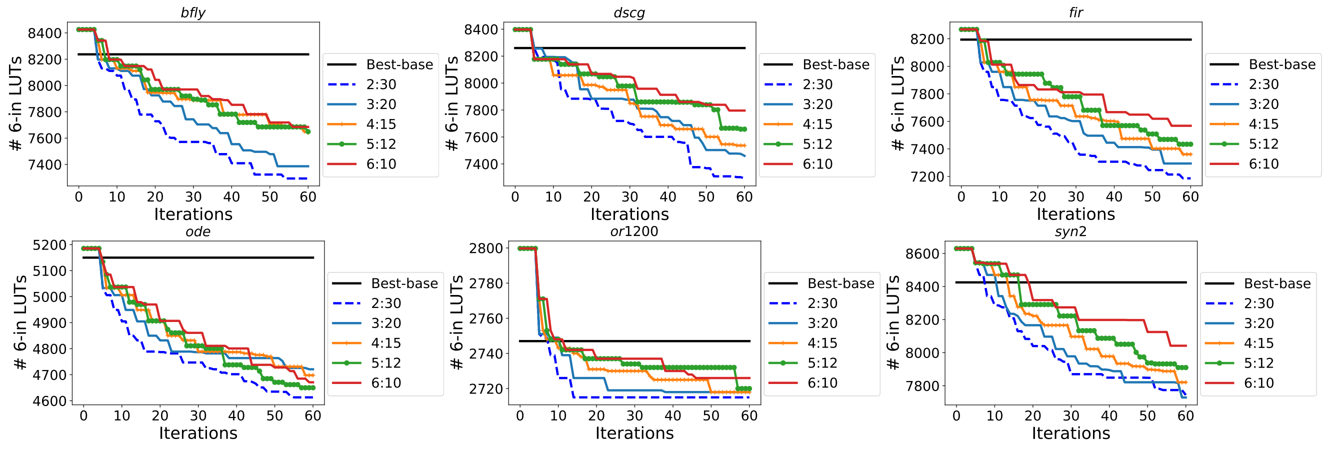

Figure 5: *FTune* in LUT minimization for FPGA technology (6-in LUT devices). `Best-base` is the best baseline result obtained from hand-crafted scripts and ML-based approaches [4, 5]; *s*:*m* (e.g., 2:30) shows the # stages verses # iterations per stage.

(Section 4.2), CNF (clauses) minimization (Section 4.3), and post-route evaluation in Xilinx Vivado (Section 4.4).

4.1 Evaluation of Logic Minimization

Objectives for FTune: **a)** minimizing the number of AIG nodes (Figure 4); and **b)** minimizing the number of 6-in LUTs for FPGA technology mapping (Figure 5). The benchmarks are listed in Table 2. **Baselines**: Baseline results are collected using the hand-crafted scripted in ABC, and results produced using ML-based approaches proposed in [4, 5]. **Specifically, for the hand-crafted baseline, we applied hand-crafted flow `resyn` 25 times on all designs.** In Figures 4 and 5, `Best-base` is the best of these three baselines.

Table 2: Details of selected VTR benchmarks for evaluating FTune. The designs are converted into BLIF format using VTR flow.

Design	I/O	AIG	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

FTune setups: As proposed in Section 3, we can configure FTune 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 are identical (i.e., $s \cdot m$).

AIG minimization results are shown in Figure 4 and LUT minimization minimization results are shown in Figure 5. We can see that, given the designs that ABC has larger design spaces, i.e., more design optimization potentials, FTune offers more optimizations over all the baselines. Another observation is that FTune outperforms baseline at early iterations for LUT minimization. This is because the DAG-aware synthesis algorithms are not developed to be LUT-mapping aware. Most of the algorithms implemented

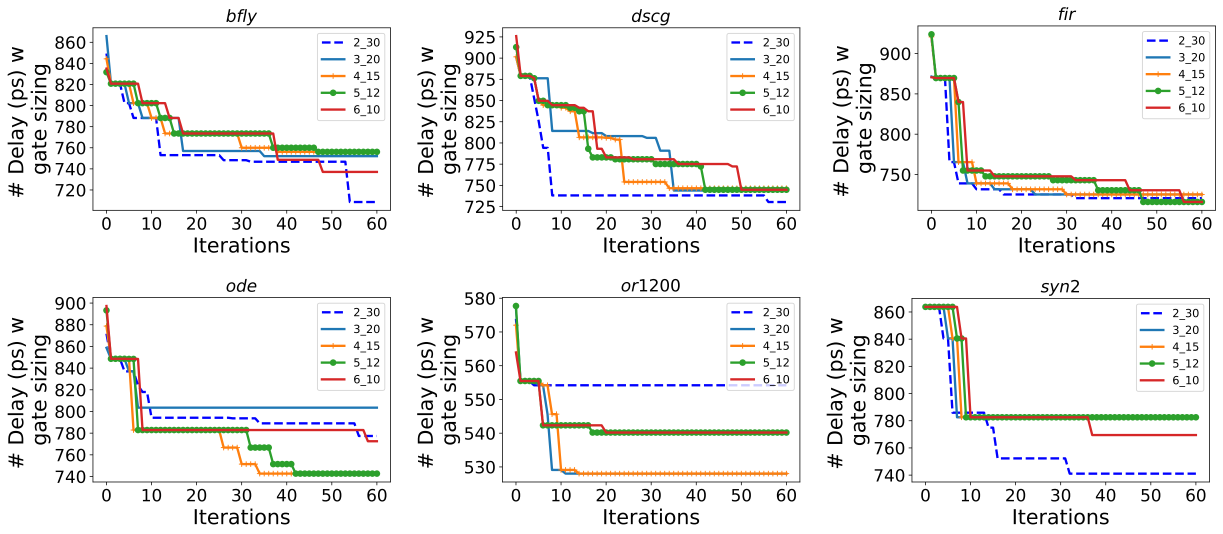

Figure 6: *FTune* in STA delay optimization w gate sizing. QoR results are obtained using Genus with ASAP 7nm library.

in ABC target minimizing the AIG nodes or the number of levels of AIG. More importantly, compared to the ML-based approaches [4, 5], for the same amount of exploration iterations, FTune is ~100 - 150× faster (runtime vary on design size), and also significantly outperforms these approach in the QoR performance.

4.2 STD Technology Mapping

Objectives for FTune: Optimize technology mapping QoRs evaluated using Cadence Genus with gate sizing, using ASAP 7nm library, targeting **a)** STA delay optimization (Figure 6); and **b)** area optimization (Figure 7). 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 FTune effectively explores the design space by finding better synthesis flows for both area and STA-delay optimization. However, FTune is not able to find any better synthesis flow after the initialization for design `or1200`. This is similar as shown in Figures 4 and 5. We believe the ABC synthesis flow design space of `or1200` is very limited for STD technology mapping.

Unlike AIG and LUT mapping minimization where FTune 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 STD delay optimization. However, for STD area minimization, we observe the same results as for AIG and LUT minimization since STD area closely correlates to AIG minimization. While these observations offer some intuition for configuring FTune for different optimization objectives, we cannot provide formal explainability at this moment.

4.3 CNF Simplification

Objective for FTune: Minimize the number of clauses in the CNF formula generated for Bounded Model Checking (BMC) with five time frames (Figure 8). **Baseline**: We compare FTune to the BMC simplification technique implemented in ABC, namely `dframes`. The results in Figure 8 are generated using *3:20* setup for FTune. For the given four BMC instances, we can see that FTune reduces averagely 21% in terms of clauses.

4.4 Case study in Vivado Flow

Finally, we evaluate FTune in an end-to-end industrial FPGA design flow (Figure 9). The input designs are two simple pipelined matrix multiplier designs generated using Vivado HLS. The baseline results are generated by running a complete design flow, i.e., Vivado HLS → Vivado. FTune results are generated as follows: 1) import Verilog generated by Vivado HLS into Yosys; 2) use the ABC interface in Yosys to call FTune for flow exploration; 3) produce Verilog using Yosys; 4) load Yosys Verilog and execute the rest of design flow using Vivado. In this integrated flow, the iterative FTune optimization process does not use information from PnR such that the runtime overhead is marginal compared to the original Vivado flow (major runtime comes from PnR). **Note that we aim to show that FTune can benefit Vivado, not proving that FTune can outperform Vivado**. Specifically, all the results in Figure 9 are generated without using any Xilinx IP (i.e., designs are implemented using LUTs only). There are two observations: First, we can see that FTune can improve ~14% post-PnR delay of both designs compared to stand-alone Vivado flow. Second, an interesting observation is that the delay distributions are significantly changed. As the technologies and FPGA architectures advance, the timing of post-PnR FPGA design is now dominated by routing delay. While using Vivado stand-alone flow, we can see that the routing delay dominates (right figures in Figure 9). However, we observe a significant delay distribution (logic delay vs. routing delay) changes that by using FTune (with ABC FPGA mapper), i.e., logic delay and routing delay are almost equivalently distributed. One possible reason is that Vivado synthesis flow has features that are PnR aware, which intends to reduce routing delay for large-scale complex FPGA designs. In contrast, the FTune setup in this case study only focuses on Boolean logic optimizations.

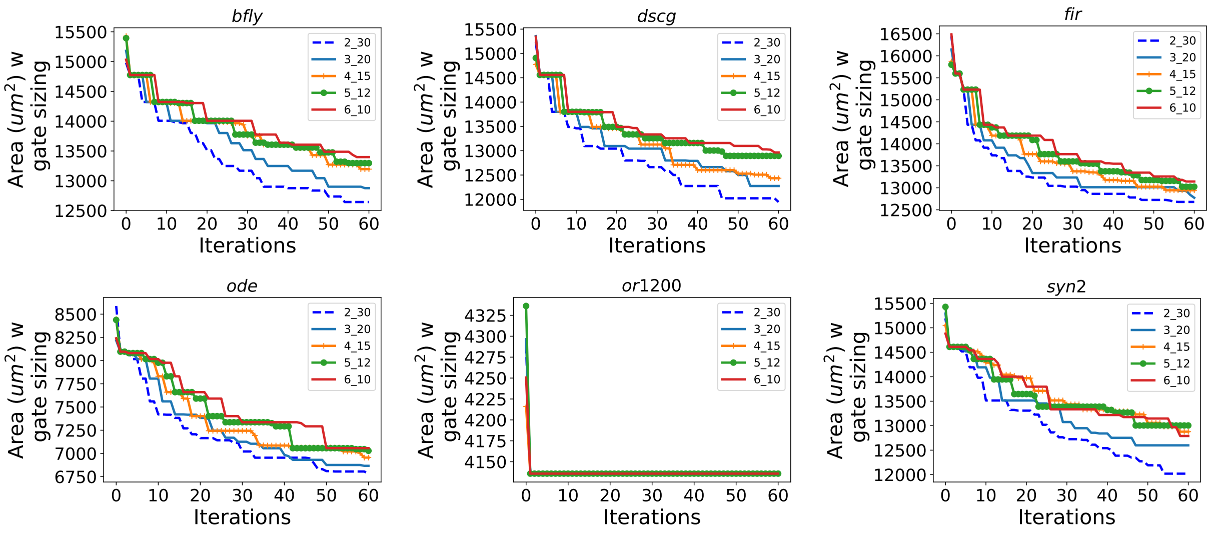

Figure 7: *FTune* in area optimization using w gate sizing. QoR results are obtained using Genus with ASAP 7nm library.

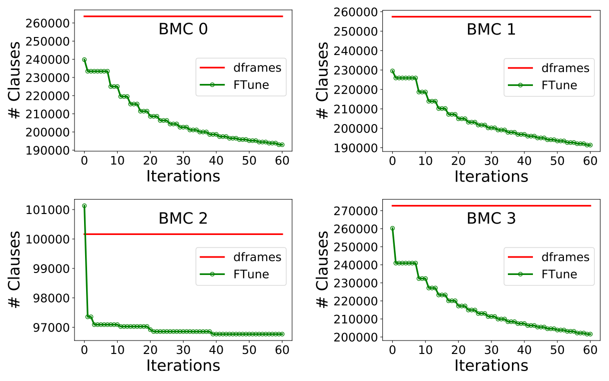

Figure 8: CNF Clauses minimization

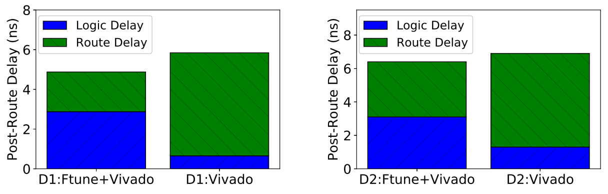

Figure 9: *FTune* case studies in Vivado using two pipelined matrix multipliers generated by Vivado HLS compiler. Timing results are collected at the post-PnR stage including logic delay and routing delay, where we observe that FTune improves the timing of both design about ~14%.

5 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, have demonstrated that FTune outperforms the existing works in both optimization performance and runtime. Future work will focus on explainability and robustness analysis of ML-based design space exploration.

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] 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.

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

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

[7] 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.

[8] C. Yu, C.-C. Huang, G.-J. Nam, M. Choudhury, V. N. Kravets, A. Sullivan, M. Ciesielski, and G. De Micheli, "End-to-end industrial study of retiming," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* IEEE, 2018, pp. 203–208.

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] H. Yang, S. Li, Y. Ma, B. Yu, and E. F. Young, "Gan-opc: Mask optimization with lithography-guided generative adversarial nets," in *DAC*, 2018.

[18] 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.

[19] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[20] 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.

[21] H. Yang, P. Pathak, F. Gennari, Y.-C. Lai, and B. Yu, "Deepattern: Layout pattern generation with transforming convolutional auto-encoder," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[22] W. Zhong, S. Hu, Y. Ma, H. Yang, X. Ma, and B. Yu, "Deep learning-driven simultaneous layout decomposition and mask optimization," 2020.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] E. Testa, M. Soeken, L. Amarù, and G. De Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *2019 56th ACM/IEEE Design Automation Conference (DAC).* IEEE, 2019, pp. 1–6.

[29] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conference.* IEEE, 2010, pp. 647–652.

[30] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning fpga compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 157–166.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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