Global Crossover: An Evolution Strategy for Logic Synthesis

Hanyu Wang
ETH, Zurich, Switzerland
hanywang@student.ethz.ch

Chang Meng
EPFL, Lausanne, Switzerland
chang.meng@epfl.ch

Giovanni De Micheli
EPFL, Lausanne, Switzerland
giovanni.demicheli@epfl.ch

***Abstract*—Technology-independent logic synthesis is crucial in electronic design automation to find the simplest logic network for the given functionality. By minimizing nodes and levels of the network, logic synthesis reduces the power consumption and propagation delay of the circuit. However, due to the inherent complexity, synthesis algorithms are mostly heuristic and may get trapped by the local optima. This paper introduces a novel design space exploration strategy to approach the global optimum more effectively. Instead of focusing on iteratively refining a single network, our approach combines multiple candidate networks simultaneously and identifies the promising "crossover". Our method assists existing strategies in achieving better circuit qualities using fewer exploration efforts. Experimental results demonstrate that our flow decreases the network size by 1.7% on average, compared with the latest design space exploration flow, within the same time limit.**

I. Introduction

Technology-independent logic synthesis finds the most efficient circuit that implements a given functionality, which is a fundamental problem for integrated circuit designers, complexity theory studies [1], and cryptography applications [2]. Reducing number of nodes in the AND-Inverter Graph (AIG) representation, for instance, is crucial for minimizing power consumption and wafer area assumption of the CMOS circuits [3], [4].

Although exact multi-level synthesis guarantees returning the smallest network for functions with less than five inputs [5], finding the optimal network for larger functions remains an open problem [6]. Various scalable logic synthesis algorithms have been developed over decades [7], including two-level logic optimizations [8], [9], algebraic methods [10]–[13], and Boolean methods [14]–[16]. However, these heuristic algorithms will likely be stuck at a local optimum without an effective *design space exploration* (DSE) strategy [17]. Indeed, randomly rewiring in genetic algorithms [18], [19] and accepting cost growth in the simulated annealing methods [20] lead to even smaller networks.

Therefore, as the efficiency and effectiveness of synthesis algorithms approach their practical limits, the emphasis in logic synthesis has progressively shifted from developing more powerful heuristics to refining the orchestration of existing scripts. Previous studies implement various techniques to fine-tune the sequence of synthesis algorithms [21]–[24], and a recent approach introduces strategies to "undo" previous optimizations to escape local optima [25].

This paper introduces an evolution strategy named *global crossover* to assist existing flows for more efficient design space exploration. As shown in Fig. 1, the overall idea of crossover is to identify and extract different portions from multiple functionally equivalent feasible networks (parents A and B) and combine their strengths to construct more advantageous networks (offspring 1, 2, and 3). Compared to existing strategies, the advantage of crossover is the utilization of multiple networks. While previous flows deal with one single network at a time, the global crossover processes multiple networks simultaneously and combines the optimizations found in different networks into a single offspring. This improves the exploration efficiency and increases the likelihood of finding the global optimum.

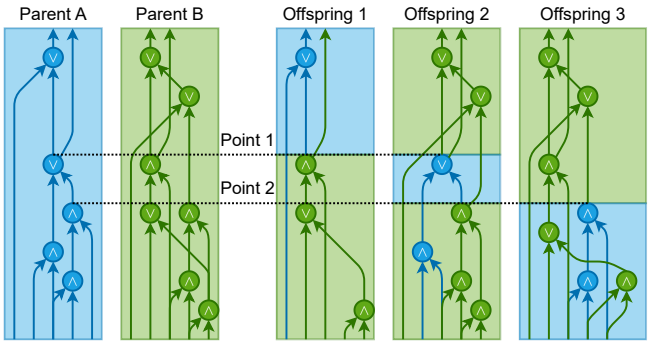


Fig. 1: Example of a logic crossover.

In the rest of this paper, we present background in Section II and motivate our work in Section III. Then, we illustrate the implementation of the global crossover in Section IV and demonstrate in Section V how this evolution strategy can be integrated into an evolutionary algorithm to improve exploration efficiency. Finally, we provide experimental results in Section VI and conclude our work in Section VII.

II. Background

A. Logic Network

Logic networks are technology-independent representations of gate-level circuits. A logic network is a directed acyclic graph where vertices represent logic gates or primary inputs, and edges represent wires. By definition, the nodes are single-output gates. For instance, *AND-Inverter Graph* (AIG) is a commonly used logic network representation where nodes are two-input `AND` gates, and the weighted edges indicate inverters.

The endpoints of a node n's incoming edges are the *fanins* of n, denoted by $\delta^-(n)$. Similarly, the set of endpoints of a node n's outgoing edges are called the *fanouts* of n, denoted

by $\delta^+(n)$. Nodes without fanins are the *primary inputs* (PIs), and nodes without fanouts are the *primary outputs* (POs).

Node f is a *support* of n if f is the immediate fanin of n or a fanout of f is the support of n. A *cone* of a node n is the set of nodes on the path between n and a set of supports of n. The *transitive-fanin cone* (TFI) of a node n is the cone between n and the set of PIs.

B. Synthesis Algorithms

Most scalable logic synthesis heuristics are based on *peephole optimization* [26]. The basic idea is to iteratively extract subgraphs, termed "windows", from the logic network and find a better replacement, referred to as a *dependency circuit*, for updates. Depending on the method used to discover the dependency circuit, synthesis algorithms can be classified into several types: rewriting [27], [28], refactoring [29], resubstitution [3], [30], and balancing [31], [32].

A common limitation across these methods is that the window size is constrained by the algorithm. For example, rewriting applies to windows with at most 6 inputs, and resubstitution finds a dependency circuit with up to 3 nodes. Beyond these thresholds, the heuristics either become non-scalable or fail to produce sufficiently effective dependency circuits. Therefore, these algorithms are regarded as local transformations when developing a synthesis flow.

C. Evolutionary Algorithm and Evolution Strategies

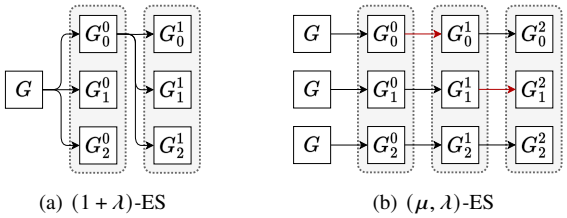
The genetic algorithm is an optimization method analogous to biology evolution [33]. The fundamental elements of a genetic algorithm are *evolution strategy* and *selection*. The evolution strategy usually includes *mutation* and *crossover*, which provides the mobility for evolution. Mutation generates offspring from a single parent, and crossover recombines multiple parents. Then, the selection is the stage to evaluate the *fitness* of individuals in a generation and choose the parents to generate the next generation. Schwefel et al. introduce the notation in Equation (1) to describe an evolution strategy.

$$(\mu/\rho^{+,}\lambda) - \text{ES}, \tag{1}$$

where μ denotes the number of parents, ρ, $\rho \leq \mu$, represents the number of parents involved when creating the offspring, and λ is the number of offspring. The plus (+) and comma (,) indicate two different rules when selecting the parents of the next generation. A *plus-selection* considers both parents and offspring in the population, while the *comma-selection* only considers the offspring.

III. Crossover: The Missing Strategy

Formulated within the framework of evolution strategies, the essence of developing a synthesis flow lies in determining the optimal values for parameters μ, λ, and ρ. This section reviews related works in the domain of logic synthesis flow designs, particularly highlighting their relationship with evolutionary algorithms. We establish that current methods provide sufficient components for an evolutionary algorithm; however, they lack a robust crossover strategy. Our motivation is to address this gap by enabling global crossover among multiple networks, thereby facilitating more versatile evolutionary strategies for logic synthesis.



(a) $(1+\lambda)$-ES (b) (μ, λ)-ES

Fig. 2: Existing evolution strategies in related works. Each square represents a logic network, and each arrow corresponds to a synthesis algorithm, acting as local transformations.

A. $(1+\lambda)$-ES and Fitness Function

The first category of synthesis flow is shown in Fig. 2a. Observing that early synthesis mistakes are often irreversible [17], these flows aim to select the most promising algorithm among all candidates. Decisions are not merely based on selecting the locally optimal network; instead, they are informed by model prediction results [23], the multi-arm bandit algorithm [21], or some hybrid heuristics [24].

From the perspective of the evolutionary algorithm, these flows apply $(1+\lambda)$-ES, generating $\lambda > 1$ offspring using one parent network. Compared with greedy algorithms that correspond to the (1+1)-ES, increasing λ improves the variety of the population. Together with a fine-tuned fitness function, these methods converge slower, explore a larger portion of the design space, and return better networks than greedy algorithms or predefined sequences of synthesis scripts.

B. Mutation and (μ, λ)-ES

Rather than focusing on selecting the correct algorithm, the second category of synthesis flow addresses the consequences of incorrect decisions by implementing two key mechanisms [25]. First, the flow introduces "decompression" algorithms that significantly restructure the network to reverse previous optimizations. Second, if the exploration gets trapped by a local optimum, the flow restarts, undoing all previous algorithms and returning to the initial network setup.

As shown in Fig. 2b, the strategy of accepting decompression algorithms can be regarded as mutations in the evolutionary algorithm. Besides, the flow would accept these offspring even if they are worse than their parents, embodying the comma-selection strategy. Furthermore, adopting the multiple restarts aligns with increasing the number of parents, essentially allowing $\mu > 1$.

C. Enabling $(\mu/\rho^{+,}\lambda)$-ES

Previous works have laid the groundwork for an effective evolutionary algorithm, yet they lack a crucial component: crossover. As illustrated in Fig. 3, current state-of-the-art flows employ (μ, λ)-ES; however, the networks are optimized separately, and the diversity within the population is not fully utilized.

This motivates our work to enable crossover for logic synthesis. The overall idea of the crossover is depicted in Fig. 1. Given two or more parent networks, the offspring of a crossover recombines the nodes from their parents to generate a functionally equivalent network. Compared with the mutation-only strategy, employing crossover could improve the design of space exploration in the following two ways.

1) Direct optimization: By combining the good properties of the parents in the offspring, the crossover may result in more advantageous networks. For example, while the two parents in Fig. 1 have sizes of 5 and 7, respectively, the first offspring created by crossover reduces the size to 4.
2) Indirect optimization: Global crossover further diminishes the likelihood of convergence to local optima leveraging the diversity in the population. For instance, offspring 2 and 3, as depicted in Fig. 1, may not immediately result in a reduced network size; however, they do integrate circuit structures previously absent in their parent networks. These newly introduced structures can potentially unveil new opportunities for optimization in subsequent synthesis algorithms.

IV. Enabling Global Crossover

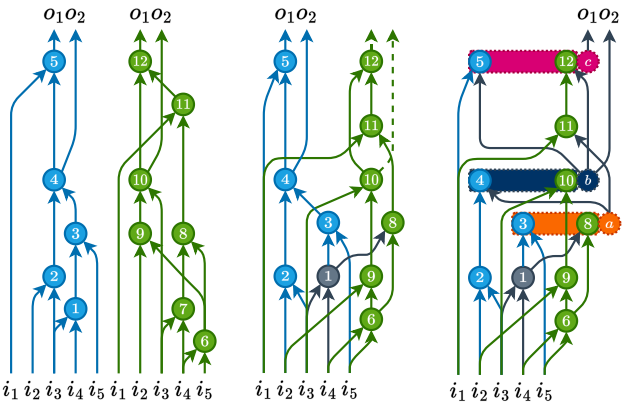
As motivated in Section III, our goal is to enable *global crossover* that generates the offspring network as a combination of multiple parent networks for logic synthesis. In the rest of this section, we will detail the problem formulation of global crossover and demonstrate the three steps to accomplish a crossover.

A. Step 1: Primary Input Alignment and Structural Hashing

Similar to the chromosome synapsis during meiosis [34], we first align the parent networks by matching their primary inputs and then merge them into a *complex network*. This alignment creates a unified basis for simulating the Boolean functions and facilitates functionality comparisons. Besides, we perform *structural hashing*, which eliminates nodes with identical immediate fanins and functionality, thereby reducing structure redundancy in the circuit. Finally, we keep only one set of primary outputs from a parent network, allowing the remaining outputs to stay dangling without any fanouts. After merging parent networks, this approach ensures that the resulting complex network maintains a streamlined structure, avoiding unnecessary complexity while preserving essential node functionalities.

Fig. 3b illustrates the resulting complex network of two parent networks in Fig. 3a after step 1. Nodes 1 and 7 in Fig. 3a are identical as they are the AND of the same inputs, i_3 and i_4; therefore, they are merged into the grey node during structural hashing.

Structural hashing is a polynomial time algorithm that improves our network representation's efficiency. As introduced in Section II, synthesis methods utilize peephole algorithms that only modify a portion of the network. Since the parent networks are generated from the same initial network using local transformations, the aggregated size of the complex network is, in practice, far less than the combined size of the individual parent networks.



(a) Initial networks (b) Complex network (c) Choice network

Fig. 3: The procedure of a global crossover.

B. Step 2: Equivalency Checking and Choice Network

After being merged into a complex network, the parent networks engage in exchanging and recombining nodes to create offspring networks. To maintain functional integrity throughout this process, it is crucial that only nodes belonging to the same *equivalence class*, a set of nodes with equivalent functionalities, can be swapped. To this end, we preprocess the complex network obtained in Step 1 and derive all functionally equivalent nodes utilizing SAT sweeping [35], [36].

Specifically, we derive a collection of simulation patterns that are capable of differentiating between varying functionalities using SAT formulation [36]. Nodes with identical simulation patterns are then subjected to formal verification methods to confirm their equivalence. Table I details the patterns extracted for the example in Fig. 3b, where the network comprises five primary inputs and eleven nodes. We use different colors to indicate different equivalence classes. For instance, the patterns observed in sets $A = \{n_3, n_8\}$, $B = \{n_4, n_{10}\}$, and $C = \{n_5, n_{12}\}$ are congruent, indicating the nodes in each set are highly likely to be functionally equivalent. In fact, they are indeed functionally equivalent after SAT checking. Therefore, they are eligible for exchange during the crossover.

To efficiently store the equivalence classes, we revisit the concept of the *choice network* [37]–[39]. Within a choice network, each equivalent class corresponds to a *choice* between

TABLE I: Matching equivalent nodes using SAT sweeping.

Pattern	i_1	i_2	i_3	i_4	i_5	n_1	n_2	n_3	n_4	n_5	n_6	n_8	n_9	n_{10}	n_{11}	n_{12}
1	0	1	1	0	1	0	1	0	1	0	0	0	0	1	1	0
2	0	0	0	1	1	0	0	0	0	1	1	0	0	0	1	1
3	0	0	1	1	1	1	0	1	1	0	1	1	0	1	0	0
4	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0
5	1	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0
6	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1

the candidate nodes. We introduce virtual nodes, also called *representatives*, corresponding to these choices; for example, nodes n_A, n_B, and n_C as illustrated in Fig. 3c. These virtual nodes act as placeholders or hubs for the collective fanouts of all nodes within a given choice. Formally, let x be a representative, then the fanout set of x is defined as the union of the fanout sets of all nodes x within its equivalence class, as shown in Equation (2).

$$\delta^+(x) := \bigcup_{y \in \mathrm{EC}(x)} \delta^+(y), \tag{2}$$

where $EC(x)$ represents the set of nodes in x's equivalence class.

The choice network model efficiently encodes potential crossover networks, where each combination of choices corresponds to a different feasible network configuration. Indeed, by employing a linear number of nodes relative to the choices, $\sum_X |X|$, a choice network can represent an exponentially growing number of network combinations, approximately $\prod_X |X|$.

C. Step 3: Global Crossover Problem Formulation

However, not all $\prod_X |X|$ network crossing over results in a valid logic network due to the necessity of adhering to the underlying graph structure of the network. To depict the feasibility region, we formulate the constraints in the crossover problem.

TABLE II: Variable declaration for the global crossover formulation.

G	(V_G, E_G)	Choice network
H	(V_H, E_H)	Offspring network
S_x	$\{0, 1\}$	Selection variable of node $x \in V_G$
D_x	$\mathbb{R}$	Depth variable of node $x \in V_G$

Table II presents the definition of related variables. Given a choice network $G = (V_G, E_G)$ constructed by steps 1 and 2 as input, our problem formulation of crossover aims to find an offspring network, $H = (V_H, E_H)$. The crossover does not create new nodes or introduce new functionality to the network but only selects existing nodes in G to generate H. Therefore, $V_H \subseteq V_G$. However, H is not necessarily a subgraph of G, as the crossover allows replacing the fanin y of a node x by another node candidate in the same equivalence class as y. This rewiring introduces node connections and network structures that do not exist in G.

In the rest of this section, we will detail the definitions in Table II and introduce the constraints to ensure the feasibility of the offspring network H.

1) Selection Variables and Depth Variables: For each node (or choice representative), x, in the choice network G, we assign two variables to capture essential characteristics of the network, the size and depth, at node x.

The *selection variable*, denoted as S_x, where $S_x \in \{0, 1\}$, is a binary variable indicating whether node x is included in the crossover network H. Assigning $S_x = 0$ excludes node x from the resulting network. Therefore, the size of the network H can be quantified by the sum of all selection variables:, i.e.,

$$|V_H| = \sum_x S_x. \tag{3}$$

The *depth variable*, denoted by D_x, where $D_x \in \mathbb{R}$, represents the depth at x. Primary inputs have zero depth, and the overall network depth, denoted by D, is the maximum depth of its primary outputs, i.e.,

$$D = \max_{x \in \mathrm{PO}} D_x. \tag{4}$$

2) Dependency and Functionality Constraints: If a node x is included in H, then all its input nodes (immediate fanins) must also be included. We formulate these requirements using *dependency constraints*, as shown in Equation (5), to maintain the logic flow from inputs to outputs.

$$\bigwedge_{x \in V_G} \bigwedge_{y \in \delta^-(x)} (S_y \vee \overline{S_x}) = 1. \tag{5}$$

This expression asserts that for every node x in the graph G, and for each fanin y of x, $S_x = 1$ implies $S_y = 1$. This ensures that all nodes in H are supported by the primary inputs.

The *functionality constraints* states that H fully implement all primary outputs from the parent networks, given by:

$$\bigwedge_{x \in \mathrm{PO}} S_x = 1, \tag{6}$$

where x can be either a node or a choice representative.

For example, this equation requires $S_b = S_c = 1$ for the choice network in Fig. 3c, where b and c are the representatives of two equivalence classes, $B = \{n_4, n_{10}\}$ and $C = \{n_5, n_{12}\}$, and correspond to primary outputs o_2 and o_1, respectively.

3) Choice Decision Constraints: During the crossover process, each choice representative can arbitrarily choose among the candidate nodes in its equivalence class gathered in Section IV-B. We use *choice decision constraints*, as shown in Equation (7), to express this procedure.

$$\bigwedge_{x \in V_G} \left[\left(\bigvee_{y \in \mathrm{EC}(x)} S_y \right) \vee \overline{S_x} \right] = 1, \tag{7}$$

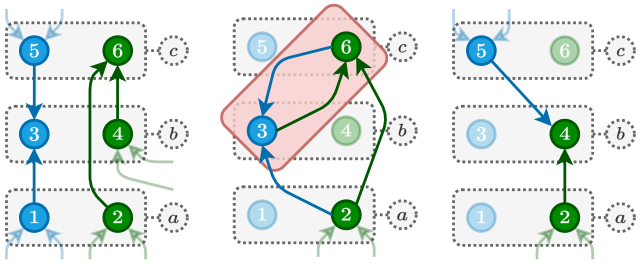
where $\mathrm{EC}(x)$ denotes the set of nodes with the same function as x. Notably, if x is not a choice representative, $\mathrm{EC}(x) = x$ and $y = x$, making the constraint a tautology for this node.

Compared to Equation (5), a choice representative x has more flexibility in propagating the dependency constraint since $S_x = 1$ implies that at least one node in $\mathrm{EC}(x)$ needs to be included by H.

For example, the choice decision constraint at representative $x = c$ corresponds to the term:

$$(S_5 \vee S_{12}) \vee \overline{S_c} = 1,$$

since n_5 and n_{12} are in the equivalence class $\mathrm{EC}(c)$. As a result, although PO constraints require $S_c = 1$, Equation (7) holds provided that $S_5 \vee S_{12} = 1$. However, dependency



(a) Complex network (b) Invalid crossover (c) Valid crossover

Fig. 4: Example of topology conflicts. Nodes $\{n_1, n_3, n_5\}$ in blue and $\{n_2, n_4, n_6\}$ in green are from different parents. Node n_3 and n_6 are XOR nodes, i.e., $n_3 = n_1 \oplus n_5$ and $n_6 = n_2 \oplus n_4$. Arbitrary choice decisions, e.g. $S_3 = S_6 = 1$, can lead to cyclic dependencies, as highlighted in the red box.

constraint for a regular node, e.g., n_{12}, corresponds to the two terms:

$$(S_{11} \vee \overline{S_{12}}) \wedge (S_b \vee \overline{S_{12}}) = (S_{11} \wedge S_b) \vee \overline{S_{12}},$$

which requires both fanins' selection variables, S_{11} and S_b to be true, if $S_{12} = 1$.

4) Depth Propagation and Topology Constraints: The crossover involves more than arbitrary choice decisions when considering the logic network's acyclic characteristic. Despite having identical functionalities, recombining nodes with different topological arrangements can lead to configurations with cyclic dependencies, which are not permissible in logic networks.

For instance, the complex network in Fig. 4a has three choices, and each has two candidate nodes from both parents. n_3 and n_6 are XOR nodes with different topologies.

$$n_3 = n_1 \oplus n_5 \text{ and } n_6 = n_2 \oplus n_4.$$

In this scenario, selecting nodes based on choices at b and c could lead to conflicts. Specifically, if $S_3 = S_6 = 1$ and $S_4 = S_5 = 0$, then the crossover, as shown in Fig. 4b, becomes cyclic and thus invalid. To prevent such conflicts and maintain the acyclic nature of the network, we implement two constraints.

First, we use *depth propagation constraints*, as shown in Equation (8), to express the depth relationship between a node x and its fanin y.

$$D_y + 1 \leq D_x, \forall x \in V_G, \forall y \in \delta^-(n). \tag{8}$$

This equation asserts the topological order of the network: if a node y is in x's transitive fanin cone, then D_y is strictly lower than D_x, i.e.,

$$y \in \text{TFI}(x) \Rightarrow D_y < D_x. \tag{9}$$

Second, we introduce *topology constraints*, as shown in Equation (10), to associate choice decision with the depth propagation.

$$D_y \leq (1 - S_y) \cdot |G| + D_x, \forall x \in V_G, \forall y \in \text{EC}(x), \tag{10}$$

where $|G| > 0$ is the size of the complex network. When $S_y = 1$, then the inequality becomes $D_y \leq D_x$. These constraints set the lower bound of $\text{EC}(x)$ as the maximal depth of selected nodes in its equivalence class. When $S_y = 0$, Equation (10) holds regardless of D_x because $D_y \leq |G|$. Therefore, depth variables of non-selected candidates do not affect D_x.

The topology constraints avoid loops during crossover by checking if the choice representative, x, of a node y is in the transitive fanin cone, $\text{TFI}(y)$. Note that this technique is frequently used in scheduling problems [40]. Leiserson et al. have proved the completeness and soundness of similar constraints for retiming [41]. In this paper, we specialize the proof for choice networks.

Proposition 1. *H is acyclic* if and only if *Equation* (8) *and Equation* (10) *hold.*

Proof. ($\Rightarrow$) holds because we can assign D_x as the index of x in the topological order given an acyclic graph, which satisfies Equation (5) by definition. Besides, $|G|$ is larger than the difference between two indices in the topological order; thus, it is sufficient to relax the constraint in Equation (10) if $S_y = 0$.

To prove ($\Leftarrow$), we assume for the sake of deriving a contradiction that there exists a cyclic graph H, whose corresponding S_x configuration satisfies Equation (8) and Equation (10).

Such a loop must contain at least one choice representative. Otherwise, it implies that one of the parent networks is cyclic, which contradicts the assumption that parent networks are valid logic networks. Let x be the choice representative on the loop, and y be the selected candidate node in $\text{EC}(x)$, i.e., $S_y = 1$. Observe that x must be in the transitive fanin cone of y to form a loop. According to Equation (9), $D_x < D_y$ holds. Meanwhile, our assumption that Equation (9) holds implies $D_y \geq D_x$ if $S_y = 1$. ↯ □

5) Objective Function for Exact Crossover: Size and depth are the two most frequently used cost metrics for logic synthesis and are derived using Equation (3) and Equation (4), respectively. We develop the objective function expressed in Equation (11).

$$\min . \sum_{x \in V_G} S_x + \alpha \cdot D, \tag{11}$$

where α decides the tradeoff between size and depth optimization. In this paper, we prioritize size optimization and set $\alpha \to 0$. Nevertheless, note that the constraints introduced in previous sections are cost-generic and independent from the objective function and the logic network representation. Besides, Wang et al. have demonstrated the capability of expressing specialized cost functions on technology-independent logic networks [42]. Therefore, we conclude that our formulation is generalizable for technology-dependent tasks.

V. Adaptive Evolutionary Algorithm

Utilizing the crossover method introduced in Section IV and various synthesis algorithms as mutation methods, we propose an evolution algorithm that equips $(\mu/\rho, \lambda)$-ES. In this section, we introduce an adaptive evolutionary algorithm framework to

orchestrate these methods and determine the hyperparameters of the evolutionary process dynamically.

An overview of the adaptive algorithm is depicted in Fig. 5. We use a state transition diagram to illustrate the dynamic decision-making on the fly. The process starts from state S_0, where the users specify the initial network and a set of available synthesis algorithms. Subsequently, we generate the first generation by randomly applying these algorithms as local moves on the input network, as shown in state S_1.

From S_1, our workflow iteratively applies evolution strategies, including mutation and crossover, to evolve the population and loop back to S_1, similarly to the typical evolutionary algorithm. Distinctively, our approach selects between three predefined sets of hyperparameters based on the *converge rate*, denoted as Δ, $0 \leq \Delta \leq 1$. We calculate the converge rate as the number of best results updates in a certain interval. For example, $\Delta = 1$ indicates that a smaller network is found and the best result is updated in every generation. Conversely, $\Delta = 0$ when the best cost remains unchanged and stuck at a local optimum.

We establish two thresholds, $\Delta_{\min}$ and $\Delta_{\max}$, and categorize the evolution strategy according to Δ's value.

- If $\Delta > \Delta_{\max}$, the network's size decreases continuously, and the best size is updated frequently. This happens if the networks possess rich optimization opportunities and are far from optimal. In this scenario, our algorithm decreases the mutation rate to focus synthesis algorithms on size reduction, as depicted by S_3 in Fig. 5. Moreover, we set the objective function in Equation (11) and employ exact crossover to minimize network size, thereby integrating optimizations found by different synthesis algorithms in the same offspring to accelerate the convergence.
- If $\Delta_{\min} \leq \Delta \leq \Delta_{\max}$, our adaptive algorithm transitions the state from S_1 to S_2 in Fig. 5 to continue optimization while preserving population diversity. Unlike the exact crossover in S_3, the crossover in S_2 randomly returns multiple suboptimal solutions in the feasibility region using heuristics. Besides, a higher mutation rate is used to facilitate escaping local optima.
- If $\Delta < \Delta_{\min}$, then the population has rapidly converged to local optima. In this case, both mutation and crossover cannot easily create new offspring. Therefore, we transition to state S_1 and reinitialize the population from the input network to recover the variety.

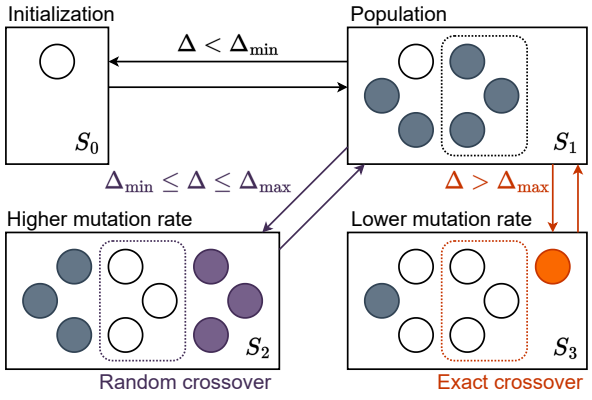


Fig. 5: State transition diagram of our adaptive evolutionary algorithm. Circles represent logic networks. We use three colors to indicate different evolution strategies selected according to the adaptive algorithm based on the converge rate Δ.

Finally, if the elapsed time reaches the user-specified time limit, the evolution process is terminated, and we return the best network discovered during the entire process.

VI. Evaluation

This section presents experimental results to demonstrate the advantage of crossover and the effectiveness of our adaptive evolutionary algorithm. In these experiments, we compare our method with a state-of-the-art design space exploration flow [25] and a flow-tuning algorithm based on multi-arm bandit models [21]. They correspond to the two evolution strategies: (μ, λ)-ES and $(1 + \lambda)$-ES, respectively.

We run these flows on the EPFL benchmark suite that comprises combinational arithmetic and control logics with different sizes [43]. All three methods aim to optimize the AIG size. Our initial AIGs are highly-optimized `LUT6` networks. We excluded one benchmark "hyp" from the benchmark suite in our experiments due to the excessively large network size, which prevents our method from completing sufficient numbers of generations within the time limit of 3600 seconds. They are first converted into AIG and then preprocessed using three iterations of ABC `compress2rs` script [44]. We convert the Boolean equality constraints to linear inequalities and utilize Gurobi [45] to solve the exact crossover formulated in Section IV. We run all the experiments on a computer with a 3.7GHz AMD Ryzen 9 5900X processor with 64GB RAM.

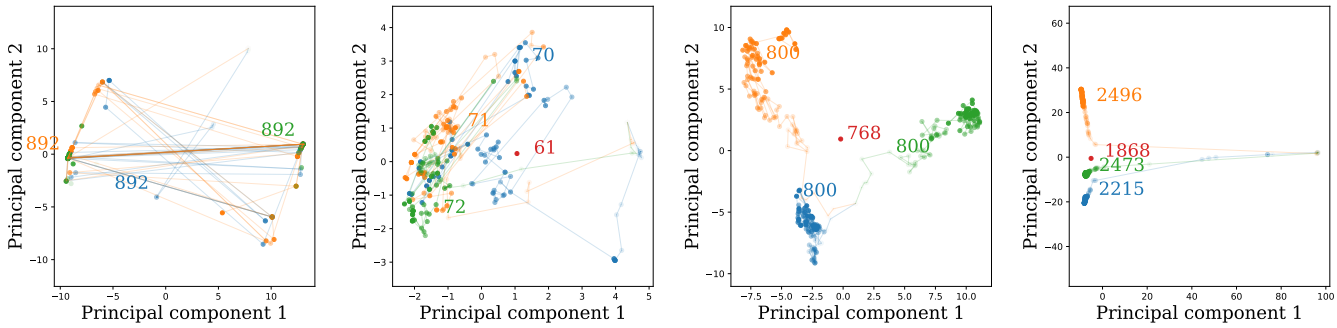
To ensure a fair comparison, all the methods employ the same set of synthesis algorithms from ABC as the local transformation [44]. `compress2rs` and `dc2` are predefined sequences of size optimization scripts and the scripts `if;mfs;fx;st` restructure the AIG network, potentially increasing the size, to escape local optima. For additional parameter setups used in our experiments, refer to Table III.

TABLE III: Parameter setup for the experiments. The upper half of parameters are common in both our method and the state-of-the-art flow [25]. The lower half is for crossover and our adaptive algorithm.

Num. restarts (num. population, μ)	4
Time limit	3600s (900s for each restart)
Compression (mutation)	`compress2rs` or `dc2`
Decompression (mutation)	`if;mfs;fx;st`
Mutation rate	S_2 (High): 100%, S_3 (Low): 50%
Num. parents in crossover (ρ)	4
Num. offspring (λ)	S_2 (Random): 4 S_3 (Exact): 1
Conv. rate thresholds	$\Delta_{\min} = 0.05$, $\Delta_{\max} = 0.5$

A. Exploration Trajectories

To demonstrate the advantages of incorporating crossover in the evolution strategy, we plot the exploration trajectories of the state-of-the-art flow in Fig. 6. Dots in the figure are



(a) Adder (b) Ctrl (c) Arbiter (optimized) (d) Arbiter (unoptimized)

Fig. 6: Design space exploration trajectories without crossover. Each dot in the graph represents an explored network. We use orange, blue, and green colors to depict three different restarts from the input network. Each trajectory comprises 100 line segments representing the first 100 local transformations. The red dot represents the network reached in Table IV with crossover. The numbers in the figure are the best networks reached in the corresponding trajectories. The first three benchmarks are highly optimized AIGs with different functionalities. The last two benchmarks have the same functionality but from different optimization stages.

explored networks, and lines represent the local transformations. The coordinates of these logic networks are determined by a $\{0,1\}$ vector composed of node selection variables S_n introduced in Section IV. The coordinate of a network is derived in the following way:

1) First, we apply a global crossover to all the explored networks and generate a large complex network G comprising all the nodes and functionalities.
2) Then, the nodes in a network H that occur in the trajectory are the subset of the vertices in G. Thus, H can be encoded as a $\{0,1\}$ vector S of length $|G|$ as demonstrated in Section IV.
3) Finally, to visualize them in a figure, we apply principal component analysis and select a two-dimensional projection that differentiates these vectors the most.

This encoding method allows us to analyze the structural difference between networks with the same size and depth. For instance, the trajectory in Fig. 6a shows the perturbation of the adder's logic network at the local optimum with a constant size of 892.

The results in Fig. 6c and Fig. 6d demonstrate the advantages of crossover in the evolutionary algorithm. The trajectories of different restarts (line segments with different colors) diverge gradually after the first few local transformations, which aligns with the observation in [17]. Instead of predicting the "correct" direction or increasing the number of restarts, crossing over these different networks is a more efficient strategy to escape the local optima. Meanwhile, mutation-only exploration relies merely on extending the length of local transformations; therefore, they are trapped by the three local optima in the solution space and fail to achieve the same quality of results as the red point.

Besides, the crossover can also assist the previous synthesis flow by combining and utilizing the strengths of different parents. This scenario is demonstrated in Fig. 6b, where the three trajectories overlap, indicating that local transformations have sufficient reachability in the solution space. In this case, increasing the number of restarts explores the same portion of the solution space, thus, does not improve the exploration efficiency of the mutation-only flow [25]. However, after enabling crossover, our evolutionary algorithm utilizes the diversity in the population and merges the advantageous substructures from different trajectories. Consequently, crossover improves the convergence rate, allowing our flow to reach smaller networks in the same time limit, as shown in Section VI-B.

Moreover, the difference between four sets of trajectories in Fig. 6 justifies our adaptive algorithm. Indeed, the solution space differs across not only benchmarks but also the optimization stage of the same benchmark, as demonstrated in Fig. 6c and Fig. 6d. Only by adjusting the parameters dynamically according to the landscape of the solution space during the exploration can we find the most appropriate strategy to improve the exploration efficiency.

B. Design Space Exploration Efficiency

Table IV demonstrates the comprehensive comparison between our flow and the state-of-the-art exploration flow [25] and a flow-tuning algorithm based on multi-arm bandit models [21]. We highlight the best size for each benchmark. Note that the performance of these flows depends heavily on the random number generator. To mitigate randomness, we report the smallest AIG sizes observed across six trials, each with a distinct random seed. We set a sufficiently large time limit for both methods and recorded the time elapsed when first achieving the displayed circuit size in the exploration, denoted as T.

We observe that the flow-tuning approach is not suitable for these benchmarks. Note that the initial networks in Table IV are highly optimized AIGs mapped from the smallest LUT-6 network and inherently hard to optimize further. On these networks, most local transformations cannot lead to immediate size reduction, and a positive reward can be achieved only by accepting these moves with negative gain. In this scenario, model-based methods must reach a sufficient deep sampling stage in order to distinguish a good flow from a bad one. However, this requires a large number of MAB iterations,

TABLE IV: Comparison with the state-of-the-art design space exploration flow [25] and a flow-tuning algorithm based on multi-arm bandit models [21]. The columns labeled "T" denote the time elapsed upon achieving the optimal size for the first time. We monitor the time limit following each local transformation, resulting in a potential slight deviation from the 3600-second time limit. The columns titled "#Gen" denote the number of generations, which correspond to the length of the synthesis algorithm sequence originating from the initial networks and indicate the depth of exploration.

	Initial				$(1+\lambda)$-ES [21]			(μ,λ)-ES [25]			$(\mu/\rho,\lambda)$-ES (Ours)		
Benchmark	Size	Depth	#PIs	#POs	Size	T(s)	#Iter	Size	T(s)	#Gen	Size	T(s)	#Gen
adder	908	322	256	129	892	3600.0	254	892	0.5	5490	892	0.5	237
bar	2688	14	135	128	2688	3600.0	60	2688	0.0	0	2688	0.0	47
div	20377	5601	128	128	20238	3600.0	2	**18818**	445.7	50	18840	3351.0	24
log2	41984	881	32	32	41984	3600.0	0	32928	1765.3	21	**32217**	2891.0	16
max	3229	448	512	130	3049	3600.0	56	3004	798.9	1323	**2990**	3630.8	189
multiplier	50147	822	128	128	38275	3600.0	2	33843	3701.7	16	**32473**	3612.6	15
sin	16425	362	24	25	10065	3600.0	7	8245	3947.7	13	**7884**	3673.7	9
sqrt	30968	6162	128	64	23049	3600.0	2	23147	2653.5	44	**22852**	3283.7	19
square	20623	701	64	128	18266	3600.0	4	16388	887.2	33	**16346**	3613.4	30
arbiter	805	312	256	129	805	3600.0	244	790	3080.6	3654	**764**	3603.8	378
cavlc	1085	37	10	11	965	3600.0	102	**403**	1358.7	3539	474	3115.1	507
ctrl	83	11	7	26	74	3600.0	1323	66	911.8	38255	**61**	3322.2	3990
dec	379	5	8	256	321	3600.0	374	322	1804.2	8852	**304**	20.0	890
i2c	1373	41	147	142	780	3600.0	208	685	1718.5	6136	**672**	2102.3	673
int2float	322	34	11	7	253	3600.0	459	128	2948.2	18809	128	1010.2	1852
mem_ctrl	7964	39	1204	1231	7051	3600.0	14	**6715**	853.8	308	6730	3075.8	182
priority	455	55	128	8	405	3600.0	343	374	1801.7	7448	**367**	346.3	868
router	100	16	60	30	98	3600.0	1086	96	1.8	30051	**94**	44.0	3426
voter	18398	128	1001	1	12224	3600.0	3	8967	834.1	122	**8932**	3597.7	86
Average	11490				9552			8342			**8195**		

which is not feasible within the time limit. This limitation also applies to other machine learning-based flow-tuning methods.

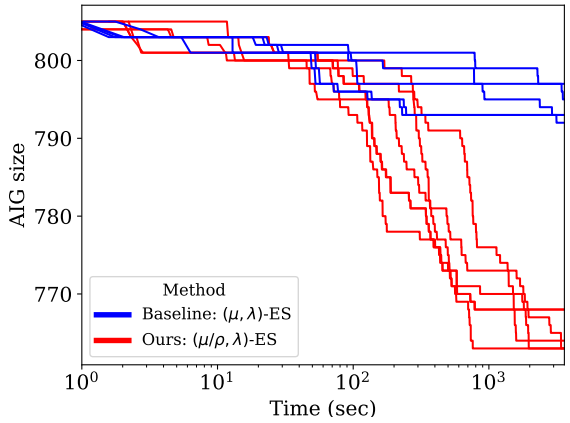


Fig. 7: AIG size optimization of the arbiter circuit. The x-axis is the CPU time in a log scale, and the y-axis displays the best AIG size achieved within the time window. We plot the results of six runs for each method. These runs have different random seeds.

Compared with the (μ,λ)-ES flow, our flow returns smaller networks within the same time limit and reduces the number of nodes by 1.7%. For example, the size reduction of the arbiter circuit from 805 to 794 takes the (μ,λ)-ES 50 minutes and 3539 iterations of compression and decompression. By employing the crossover strategy, our method reduces the circuit size further down to 764, which is a significant improvement considering the limited optimization opportunities in these benchmarks. The process of size optimization is displayed in Fig. 7. Although the trajectories are affected by the random seed, the trend is clear that our flow converges faster and achieves better results.

Note that crossover does not always improve the exploration efficiency and does not necessarily lead to better circuit size, for example, on the benchmark "cavlc" in Table IV. Our flow finds a network with 474 nodes, while the network size found by the baseline is 403. This is because of the runtime overhead of employing crossover. As displayed in Table IV, the number of generations of our flow is significantly lower than the baseline within the same time limit. In the one-hour time limit, our flow accomplishes 507 generations when optimizing "cavlc" while the baseline flow finishes 3539 iterations. Therefore, the maximum distance restricts our reachability in the solution space.

VII. Conclusion

Over decades, various logic synthesis heuristics have been developed, demonstrating sufficient effectiveness in generating local transformations. Nowadays, the design space exploration strategies that orchestrate these local transformations are becoming the bottleneck of synthesis flows. This paper proposes global crossover, an evolution strategy for logic synthesis, which identifies and extracts different portions from multiple feasible networks and combines their strength to construct a more advantageous network. Compared with mutation-only flows, crossing over multiple networks escapes the local optimum more easily while allowing a faster convergence rate. We employ the proposed crossover in an adaptive evolutionary algorithm to adjust the strategies on the fly. Experimental results show that our flow achieves better circuit qualities using fewer exploration efforts.

Acknowledgement

Many thanks to Dr. Siang-Yun Lee, Andrea Costamagna, and Alessandro Tempia Calvino for the helpful discussion. This work is supported in part by Synopsys Inc.

References

[1] V. Kabanets and J.-Y. Cai, "Circuit minimization problem," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 73–79.

[2] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo random bits," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 227–240.

[3] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *International Workshop on Logic Synthesis*, vol. 6, 2006, pp. 15–22.

[4] Y. Miyasaka, "Transduction method for AIG minimization," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 398–403.

[5] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2019.

[6] R. Ilango, B. Loff, and I. C. Oliveira, "NP-hardness of circuit minimization for multi-output functions," in *35th Computational Complexity Conference (CCC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[7] E. Testa *et al.*, "Extending boolean methods for scalable logic synthesis," *IEEE Access*, vol. 8, pp. 226 828–226 844, 2020.

[8] P. McGeer, J. Sanghavi, R. Brayton, and A. S. Vincentelli, "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," in *Proceedings of the 30th international design automation conference*, 1993, pp. 618–624.

[9] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984, vol. 2.

[10] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.

[11] C. Yu, M. J. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 9, pp. 1907–1911, 2018.

[12] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2015.

[13] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot, "Multilevel synthesis minimizing the routing factor," in *27th ACM/IEEE Design Automation Conference*. IEEE, 1990, pp. 365–368.

[14] J. A. Darringer *et al.*, "LSS: A system for production logic synthesis," *IBM Journal of research and Development*, vol. 28, no. 5, pp. 537–545, 1984.

[15] S. Muroga *et al.*, "The transduction method-design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

[16] E. Testa *et al.*, "Scalable Boolean methods in a modern synthesis flow," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1643–1648.

[17] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[18] J. F. Miller, D. Job, and V. K. Vassilev, "Principles in the evolutionary design of digital circuits—part I," *Genetic programming and evolvable machines*, vol. 1, no. 1, pp. 7–35, 2000.

[19] B. W. Goldman and W. F. Punch, "Analysis of cartesian genetic programming's evolutionary mechanisms," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 359–373, 2014.

[20] P. Färm, E. Dubrova, and A. Kuehlmann, "Integrated logic synthesis using simulated annealing," in *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, 2011, pp. 407–410.

[21] C. Yu, "Flowtune: Practical multi-armed bandits in Boolean optimization," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[22] 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.

[23] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 581–586.

[24] F. Liu *et al.*, "CBTune: Contextual bandit tuning for logic synthesis," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

[25] S.-Y. Lee, H. Riener, and G. De Micheli, "Customizable on-the-fly design space exploration for logic optimization of emerging technologies," in *International Logic Synthesis Workshop (IWLS), Lausanne, Switzerland, June 5-6, 2023*, 2023.

[26] S.-Y. Lee and G. D. Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3958–3971, 2023.

[27] I. Háleček, P. Fišer, and J. Schmidt, "Towards AND/XOR balanced synthesis: Logic circuits rewriting with XOR," *Microelectronics Reliability*, vol. 81, pp. 274–286, 2018.

[28] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 429–430.

[29] R. K. Brayton, "The decomposition and factorization of Boolean expressions," 1982, pp. 49–54.

[30] H. Sawada *et al.*, "Logic synthesis for look-up table based FPGAs using functional decomposition and Boolean resubstitution," *IEICE TRANSACTIONS on Information and Systems*, vol. 80, no. 10, pp. 1017–1023, 1997.

[31] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 375–382.

[32] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[33] I. Rechenberg, "Evolution strategy: Nature's way of optimization," in *Optimization: Methods and Applications, Possibilities and Limitations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 106–126.

[34] O. Rog and A. F. Dernburg, "Chromosome pairing and synapsis during caenorhabditis elegans meiosis," *Current opinion in cell biology*, vol. 25, no. 3, pp. 349–356, 2013.

[35] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[36] L. Amarú *et al.*, "SAT-sweeping enhanced for logic synthesis," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[37] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[38] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 813–834, 1997.

[39] L. Stok, M. A. Iyer, and A. J. Sullivan, "Wavefront technology mapping," in *Proceedings of the conference on Design, automation and test in Europe*, 1999, pp. 108–es.

[40] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–32, 2021.

[41] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[42] H. Wang, S.-Y. Lee, and G. De Micheli, "AnySyn: A cost-generic logic synthesis framework with customizable cost functions," *arXiv preprint arXiv:2311.14721*, 2023.

[43] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis*, Mountain View, CA, Jun. 2015.

[44] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[45] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com