Master's Thesis of Engineering

Fast Automatic Circuit Design Framework Using Genetic and Reinforcement Learning Algorithm

유전알고리즘 및 강화학습을 사용한 고속 회로 설계 자동화 프레임워크

February 2022

Graduate School of Convergence Science and Technology
Seoul National University
Intelligence and Information Major

Jiwoo Hong

Master's Thesis of Engineering

Fast Automatic Circuit Design Framework Using Genetic and Reinforcement Learning Algorithm

유전알고리즘 및 강화학습을 사용한 고속 회로 설계 자동화 프레임워크

February 2022

Graduate School of Convergence Science and Technology
Seoul National University
Intelligence and Information Major

Jiwoo Hong

Fast Automatic Circuit Design Framework Using Genetic and Reinforcement Learning Algorithm

유전알고리즘 및 강화학습을 사용한 고속 회로 설계 자동화 프레임워크

지도교수 전 동 석

이 논문을 공학석사 학위논문으로 제출함
2022 년 1 월

서울대학교 대학원
지능정보융합학과
홍 지 우

홍지우의 공학석사 학위논문을 인준함
2022 년 1 월

위 원 장: 안 정 호 (인)

부위원장: 전 동 석 (인)

위 원: 곽 노 준 (인)

Abstract

Although design automation is a key enabler of modern large-scale digital systems, automating the transistor-level circuit design process still remains a challenge. Some recent works suggest that deep learning algorithms could be adopted to find optimal transistor dimensions in relatively small circuitry such as analog amplifiers. However, those approaches are not capable of exploring different circuit structures to meet the given design constraints. In this work, we propose an automatic circuit design framework that can generate practical circuit structures from scratch as well as optimize the size of each transistor, considering performance and reliability. We employ the framework to design level shifter circuits, and the experimental results show that the framework produces novel level shifter circuit topologies and the automatically optimized designs achieve 2.8-5.3× lower PDP than prior arts designed by human experts.

keywords: Circuit Design Automation, Deep Learning, Evolutionary Algorithm, Level Shifter, Reinforcement Learning.

student number: 2019-21653

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1. Introduction

With increasing hardware design complexity and variability of the fabrication process, design automation has been widely adopted in a large portion of the IC design process. For instance, various electronic design automation (EDA) tools are now available for designing digital blocks and SoCs (System-on-Chips). Using a standard cell library, the EDA tools can generate a large block composed of millions of logic gates very efficiently [1]. However, when it comes to designing circuits, design automation remains a challenge. Most digital and analog circuits are still carefully designed by human experts due to high design complexity and reliability concerns [2].

Circuit design automation can be decomposed into two design problems: circuit topology selection and transistor size optimization. Because the topology mainly sets the limit on the performance and reliability a circuit can achieve, it is important to choose a proper circuit topology in the first place. We also need to optimize the size of each transistor to realize its true potential. Various approaches have been reported for automatic circuit topology generation. For digital logic gates, the Boolean expression factoring method that generates series-parallel (SP) associations of transistors for the given function was suggested [3]. The authors in [4] proposed an improved graph-based method that creates a logic gate by introducing Non-Series-Parallel (NSP) arrangements into the SP structure, thus reducing the number of transistors. But these methods regard transistors as ideal switches and hence are only applicable to designing digital logic gates based on static operations.

There have also been several circuit topology synthesis approaches aimed at more general circuits. The library-based methods [5, 6] select one of the predefined circuit structures (e.g., a two-stage amplifier) in the library based on the desired operating characteristics. However, one must construct a library containing all possible circuit structures in advance, which is a time-consuming process that also necessitates a con-

siderable amount of human effort. Building-block-based methods [7–9] take a similar approach, but rely on a library of smaller building blocks such as a current mirror and a differential input pair. They employ various algorithms to search for the best topology, such as multi-objective evolutionary algorithm [7], framework for explorative analog topology synthesis method (FEATS) [8], and graph-grammar-based topology generation (GGTG) [9]. Since the library- or building-block-based approaches have relatively limited search space, they are suitable for the fast generation of integrated circuits using a well-established topology. However, the search space is constrained within the predefined set of circuit structures or building blocks, and hence they are less adaptive to changes in design parameters or fabrication process. In addition, there is little possibility that they could generate a novel topology that has not been studied yet.

On the other hand, the transistor-based methods [10–13] do not rely on predefined components for topology generation; instead, they progressively construct a circuit by adding or removing a transistor in the topology. For instance, the circuit-constructing robot (CC-BOT) [11] starts with a single node and conditionally adds a transistor following an evolutionary algorithm. An active bot moves to a newly created node and continues adding transistors from there. The algorithm in [13] represents transistors and passive devices as a 3-node graph (*hypergraph*) and an edge, respectively. In each generation, it removes and adds multiple hypergraphs and edges, also following an evolutionary algorithm. The transistor-based approaches have a significantly larger search space and, as a result, are capable of generating an optimal circuit topology under different design constraints. And these approaches do not require much prior knowledge on the target circuit, removing the need for the aid of human experts during the design process. However, they essentially rely on trial-and-errors; an inefficient search algorithm results in very slow search speed, requiring an extensive amount of SPICE simulations to evaluate candidate integrated circuits.

Transistor sizing is another crucial part of integrated circuit design automation since it directly affects the performance and reliability of a circuit. The authors in [14]

proposed the multi-objective uncertain optimization with ordinal optimization LSS and parallel computation (MOOLP) optimizer based on a differential evolutionary algorithm, whereas other prior works suggest using particle swarm optimization [15, 16] or Bayesian methods [17, 18]. Recent works demonstrate promising results by applying deep learning algorithms to transistor sizing optimization (Fig. 1.1). For instance, Learning to Design Circuit (L2DC) [19] and AutoCkt [20] adopt reinforcement learning (RL) for optimizing transistor sizes in analog amplifiers. It was demonstrated that those RL-based approaches could successfully optimize the integrated circuit to meet the given design constraints such as gain, bandwidth, and input-referred noise. While the RL-based methods achieve significantly faster convergence than conventional optimization algorithms, they still need numerous SPICE simulations during optimization. Also, both L2DC and AutoCkt utilize prior knowledge on the circuit topology during optimization (e.g., the signal path and tightly coupled transistors), limiting their applicability to other types of integrated circuits. To address these problems, the authors in [21] employ a graph convolutional network in RL (GCN-RL) and utilize transfer learning. They show that using a pre-trained network can reduce the number of SPICE simulations in optimizing two-stage and three-stage transimpedance amplifiers. However, the initial training of the neural network still requires a large number of SPICE simulations, and the pre-trained network is only effective when applied to another circuit with a similar structure.

Level shifter circuits are widely used in digital systems to convert the level of signals between voltage domains. The signals from the internal core must be boosted to communicate with external discrete components or the data between core blocks with different supply voltages should be level converted for proper operation. Various level shifter circuit topologies have been studied, and the optimal topology can vary greatly depending on the operating conditions such as voltage conversion range (e.g., core to core or core to I/O) and power budget. For instance, differential cascode voltage switch (DCVS) exhibits better conversion speed and energy efficiency for conversion between

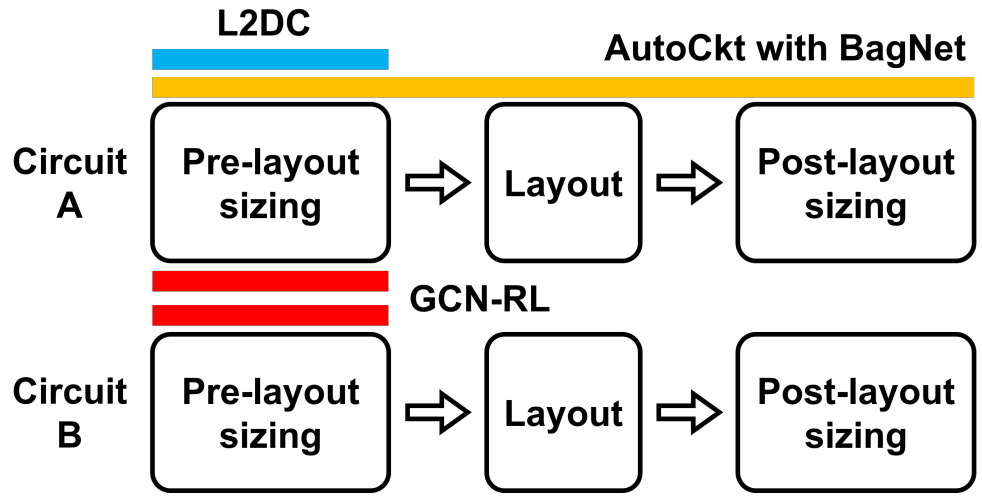


Fig. 1.1: Trend of size optimization study

core voltage domains, whereas Wilson current mirror level shifter (WCMLS) and its variant are suitable for converting subthreshold voltage input due to relaxed contention [22–24]. Therefore, we may have to switch to a totally different topology and start the design process again when the design constraints and operating conditions change, making conventional circuit design automation frameworks unsuitable for level shifter design.

In this work, we propose a unified circuit design automation framework that can generate an optimal circuit topology from scratch as well as optimize the size of each transistor. Our key contributions are: i) a 2-stage circuit design framework that significantly speeds up the design process, ii) a new voltage-based graph representation of integrated circuits, iii) a fast circuit optimizer adopting a multi-agent RL algorithm for faster convergence, and iv) a process variation-aware optimization algorithm that results in a practical, robust design. The framework was employed to design a level shifter circuit, and the resulting level shifter circuits are fabricated in a 180nm CMOS process to validate the effectiveness of the proposed circuit design framework.

The rest of the paper elaborates on the proposed framework as follows: Chapter II discusses related studies. Chapter III describes the overall architecture of the framework and its distinct features. Chapter IV discusses the experimental results, and Chapter V concludes the paper.

Chapter 2. Related work

In this chapter, we discuss the research related to the proposed framework. We will discuss the genetic algorithm related to the first step of the framework, and the algorithm that is the basis of this study. Also, we will discuss reinforcement learning related to the second step and look at each major algorithm. Lastly, We will discuss level shifter.

2.1 Genetic Algorithm

The genetic algorithm is one of the representative methods of the evolutionary algorithm and can be used in various optimization problems. Genetic algorithms that mimic natural selection of genes utilize randomness in the search process and have strengths in large search spaces or multi-modal spatial searches. In general, even when the problem to be solved is uncomputably complex, the genetic algorithm can obtain an answer close to the optimal solution. Therefore, it is widely used to solve complex nonlinear or incalculable problems in various fields.

Genetic algorithm is performed with offspring as the most basic unit. Each offspring has several genes corresponding to the metric to be optimized, and how well these genes are suitable for the final solution is evaluated through fitness value. Therefore, in order to apply the genetic algorithm, the solution must be expressed in the form of a gene, and how well the solution is suitable must be calculated through the fitness function.

The genetic algorithm repeats selection, crossover, mutation, and replacement per generation (Fig. 2.1). In the selection step, fitness value is measured and a parent pool is created by selecting offspring candidatesto be transmitted its traits to the next generation. In the crossover step, two offsprings are randomly selected from the parent

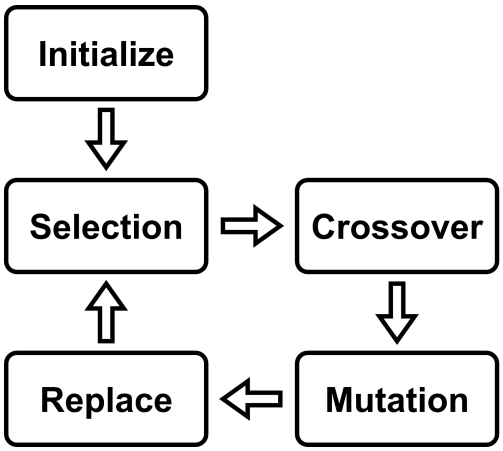


Fig. 2.1: Process of genetic algorithm

pool created in the previous step, and then the gene crossover operation is performed. In crossover process, a part of the gene is stochastically swapped with each other. The offsprings created through crossover process go through the mutation process. In the mutation process, the order or value of genes in offspring are changed stochastically. Lastly, in the replace stage, a new population is formed with newly created offsprings and passed on to the next generation. The new population passed in this way repeats the selection, crossover, mutation, and replacement process again.

2.2 NeuroEvolution of Augmenting Topologies (NEAT)

The NeuroEvolution of Augmenting Topologies (NEAT) [25] algorithm is one of the genetic algorithms proposed by Kenneth O. Stanley in 2002 and continues to be utilized and research for improvement is in progress [26–30]. The NEAT algorithm, which was proposed to develop the initial machine learning network structure and weights, applied the method of expressing the network as a graph to the genetic algorithm. Unlike the existing genetic algorithm, the NEAT algorithm shows different characteristics while graphing the network. The NEAT algorithm has two types of genes classified into connection and node. The connection gene has the innovation

number, which will be explained later, weight, in node, out node, and enable as internal properties. The node gene has the innovation number and type as internal properties. In general, unlike the existing genetic algorithm in which the number of genes is the same, the NEAT algorithm for searching a graph changes the number of two types of genes as the composition of the graph are changed.

The mutation of the NEAT algorithm has been changed from the simply changing the state of a gene like general genetic algorithm to changing the graph. The basic NEAT algorithm has three operations (changing weight, adding connection, adding node) for mutation process. Changing weight changes the weight property of the connection gene. Adding connection selects two random nodes and adds the connection gene which connect two selected nodes. Adding node creates a new node gene by selecting an existing connection gene, adding the connection gene that connects the in node to the new node and adding the connection gene that connects the out node and the new node, then disables the existing connection gene. Through these three mutation functions, the NEAT algorithm continuously increases the nodes and connections of the graph and optimizes the machine learning network structure by changing the weight.

In the NEAT algorithm, since the number of genes continuously changes, crossover is also difficult to perform in the conventional way. In order to explain crossover of NEAT algorithm, understanding innovation number is needed. In the NEAT algorithm, the connection gene and the node gene have innovation numbers in individual order. However, if each offspring has an individual innovation number, it is difficult to determine whether the created graph structure is the same or not. To solve this problem, the NEAT algorithm introduces the global innovation number and historical marker. When a new gene is created during the mutation process, historical markers are used to find out whether there is a gene that forms the same structure as the corresponding gene. If there is no gene forming the same structure in the historical marker, the new gene is added to the historical marker, the current global innovation number is assigned as

the innovation number of the gene, and the value of the global innovation number is increased by 1. All historical markers are initialized when one generation ends. In this way, all offsprings have the same innovation number for the gene of same structure that occurs in the same generation.

Unlike the conventional genetic algorithm which crossover offsprings selected by the selection process, the NEAT algorithm crossovers based on species. species is a set of offsprings whose distance result of distance function from seed offspring of species do not exceed a certain threshold, meaning that graphs with similar structures are gathered. Crossover by species can compensate for the case where achieve good performance structurally possible, but is not achieved due to weight. Also, It improve the algorithm in terms of diversity. In the NEAT algorithm, the species fitness is calculated through the fitness value of the offsprings belonging to the species, and the number of offsprings to belong to the next generation is determined for each species in proportion to this value. Species that do not score good fitness value create a small number of offsprings, so it compensates for temporary poor performance to species of lower fitness value, but gives more development opportunities to species of better fitness value and finds the direction of optimization better. When the number of offspring generation is determined, the offspring with the best performance in each species is included in the population of the next generation, and the parent pool is created by collecting the top offspring among offsprings belonging to the species. In the crossover process, based on an offspring with a high fitness score, if two offspring genes have each other at the same time, the property of the gene of base offspring is probabilistically changed. If the gene is not in the base offspring and has only the other offspring, the gene is probabilistically included to the base offspring. Through this process, a new offspring is created that will be passed on to the mutation process. This process proceeds as many as the number of offsprings created by each species, and the created offsprings are sent to the mutation process.

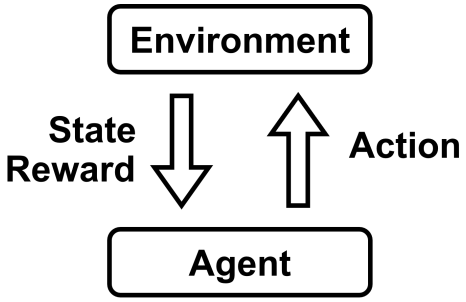


Fig. 2.2: Basic structure of RL

2.3 Reinforcement Learning (RL)

Reinforcement learning (RL), which is one of the fields of the machine learning where various studies are being conducted recently, aims to learn networks based on trial and error. RL is applied to applications that need to make decisions sequentially based on the Markov Decision Process (MDP), and most of them learn the action or sequence of actions to receive the maximum reward by using and changing the Bellman equation.

In RL, there are basically an agent that can perform a action and an environment where an agent performs an action. Fig. 2.2 shows how the environment and agent interact. The agent determines which action to perform based on the current state, and executes the corresponding action in the environment. The environment is changed based on the action performed by the agent, and then the state and reward due to the action are delivered to the agent as next state and reward. The agent uses this reward for learning.

In RL, the rules used by the agent to determine the actions to be performed are called policies. There are two types of policy: a deterministic policy that outputs one action value and a stochastic policy that outputs the probability distribution of an action

value. The difference between these two policies is the behavior of the agent. The behavior of the agent is divided into exploitation, which selects the optimal behavior, and exploration, which acquires samples by performing various behaviors. The stochastic policy selects the most optimal action in the exploitation process and performs the action, and in the exploration process, it selects a random action and performs the action. However, since deterministic policy outputs one action value, the action output in the exploitation process is used, and in the exploration process, a random noise value is added to the action to create a random action value.

The policy of RL is also closely related to the use of learning data. If the policy determining behavior and the learning policy are the same, the experience data accumulated with the current policy cannot be used to update the next policy after policy update. This type of policy is called on-policy, and on-policy reduces data efficiency because it has to discard existing experience data and accumulate new data every time it is updated. Conversely, if the policy that determines the behavior and the learning policy are different, the experience data accumulated with the current policy can be used any time regardless of the policy update. This method is called off-policy, and a space to store experience data, such as replay memory, is used, and data accumulated in the past is continuously used for learning. So data efficiency is good. Therefore, in general, it is better to use the on-policy method for applications that can easily obtain data, and use the off-policy method for applications that are difficult or take a long time to obtain data.

RL also classifies algorithms according to the presence or absence of an environment model that an agent can use when searching for an environment. The agent's access to the environment model means that the agent can accurately predict the state and reward it will acquire in the future. Therefore, it becomes possible to establish a plan by using these predictions in the process of selecting an action, and because the sample made through a good plan is used, the sample efficiency or data efficiency is greatly increased. This method is classified as model-based RL. However, in general,

environment models are often not available for use by agents. In this case, the agent continuously explores the environment to learn the environment model and uses the learned model to learn the policy, but there are cases in which bias exists in the learned model, and proper learning occurs in many cases. RL that does not use an environment model for learning is called model-free RL, and although there is a disadvantage of low data efficiency, there is an advantage of being easy to implement, so many studies are being conducted.

2.4 DDPG, D4PG, and PPO

Recently, many studies have been conducted to apply neural networks to RL. The deep deterministic policy gradient (DDPG) [31] algorithm was proposed in 2015 and is an RL algorithm using deterministic policy with model-free, off-policy, and actor-critic structure features. DDPG combines deterministic policy gradient (DPG) [32] using deterministic policy for the first time, and deep Q-network (DQN) [33], which is evaluated to use replay memory and stabilized Q-function learning, through an actor-critic structure. DDPG algorithm also can be applied to continuous action space application which is more complex problem.

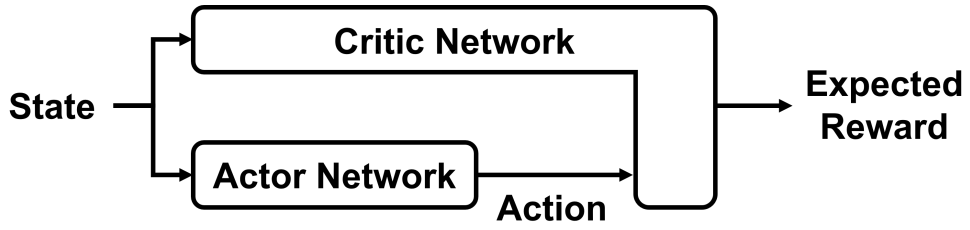


Fig. 2.3: Network structure of DDPG

The network structure of DDPG is shown in Fig. 2.3. The DDPG algorithm which has actor-critic structure is divided into actor network that receives state as input and outputs action, and critic network that receives state and action as input and outputs expected reward. Both networks are composed of neural networks and are trained through backpropagation.

The operation of DDPG is performed as follows. When the agent creates an action by inputting the current state into the actor network, the action is performed in the environment through the created action and the next state and reward are delivered. The agent creates a sample by grouping the current state, action, reward, and next state, and delivers the sample to the replay memory. In case of real-time learning, if one sample is received from the replay memory, a mini-batch is made by randomly selecting as many samples as the predefined mini-batch size. This mini-batch is used to train the network. Learning proceeds from the critic network. Backpropagation is performed using the expected reward value, which is created through the action and the current state in the sample, and the sum of the reward in the sample and the next expected reward value. the next expected reward value is created by inputting the next state into the network. After learning the critic network, the actor network is trained, and the learned critic network is used in this process. After passing the current state in the sample to the actor network and the critic network, the gradient is propagated, and the gradients for learning the actor network are propagated through the critic network. For this reason, it is very important to train the critic network well in DDPG.

The distributed distributional deep deterministic policy gradient (D4PG) [34] algorithm is a RL algorithm derived from DDPG. While the existing DDPG algorithm used a scalar value critic network in which the expected reward value is expressed as a scalar value, the D4PG uses a distributional critic network in which the expected reward value is expressed as a distributional form(Fig. 2.4). It can be seen that the intrinsic uncertainty generated in the process of approximating the function is better expressed in the distributional critic network, so that the learning is performed more

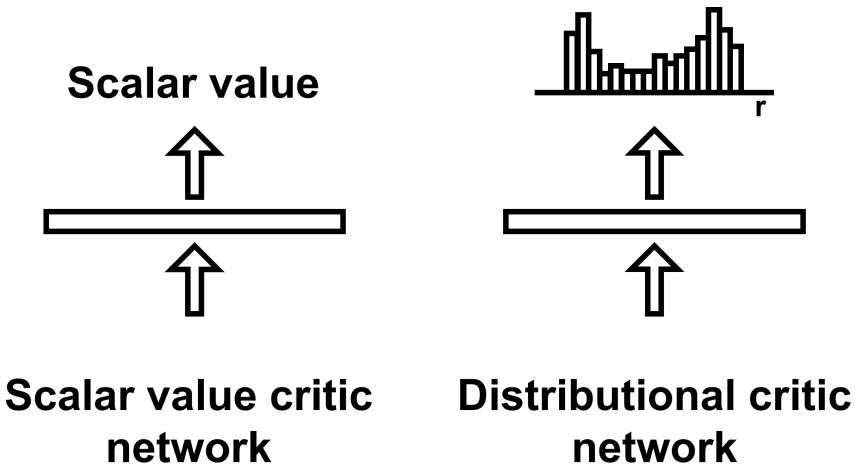


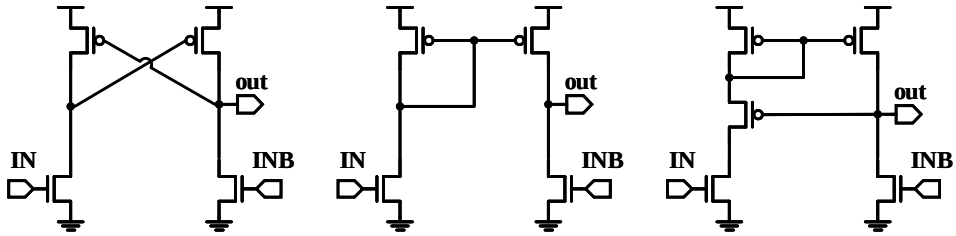
Fig. 2.4: Difference type of critic network

stably.

Proximal policy optimization (PPO) [35] algorithm is a policy optimization RL algorithm with model-free, on-policy, and multi-agent features. To improve training stability, PPO uses a method that imposes restrictions on the policy update size. The operation of the PPO proceeds as follows. After the N-agent collects samples during T-step, K mini-batches are made using the collected sample data. If you update K times through K mini-batches created in this way, all data accumulated in the past are discarded because PPO is on-policy. Repeat this process over and over again. PPO shows good performance in a simple way on various benchmarks.

2.5 Level Shifter

As described in the introduction chapter, the level shifter is used when the voltage levels of the signals of the two circuit parts are different, such as core to core or core to I/O. As for the level shifter, there are a level shifter with the same vss before and after level conversion and a floating level shifter with a floating node as vss after conversion. While the former is used in general VLSI circuits, the latter is mainly used in high



(a) DCVS (b) current mirror (c) WCMLS

Fig. 2.5: Conventional level shifters

voltage applications. In this study, we intend to discuss a level shifter with the same vss before and after level transformation.

Various level shifters are being studied, but conventionally used level shifters are DCVS, current mirror, and WCMLS. DCVS is a level shifter that applies positive feedback to P-channel MOSFET of differential structure as shown in Fig. 2.5(a). DCVS with a structure in which the signal changes when the power of N-channel MOSFET to take current out is stronger than the power of P-channel MOSFET to put current into the net is strong against noise and has fast switching speed due to positive feedback. but it is not easy to operate at near sub-threshold voltage. As a way to solve this problem, various studies have been conducted [36, 37]. Level shifter in [36] uses an auxiliary circuit that limits the current in the conversion process. In [37], the number of transistors is reduced by changing the auxiliary circuit to a simple P-channel MOSFET structure.

The current mirror level shifter is a level shifter using the P-channel MOSFET of the simplest type of current mirror structure as shown in Fig. 2.5(b). When the input becomes V_{DDL}, the voltage level of the output net rises as the current flows. The current mirror level shifter has the advantage of being able to operate at low voltage because there is no contention between P-channel MOSFET and N-channel MOSFET, but there is a disadvantage that static current flows continuously when the input is V_{DDL}. To solve this problem, various studies have been conducted [22, 23]. WCMLS

(Fig. 2.5(c)) added P-channel MOSFET gated by output to the P-channel MOSFET drain part of the current mirror, and in [23], a current mirror in the opposite direction was added above the P-channel MOSFET to prevent static current.

Chapter 3. Proposed circuit design framework

The overall flow of the proposed circuit design framework is shown in Fig. 3.1. Instead of relying on a single algorithm to design a circuit, we propose to split the design process into two distinct stages. The first stage (*topology generator*) employs an genetic algorithm to search for candidate circuit structures quickly. The second stage (*circuit optimizer*) performs an RL-based transistor size optimization on the generated integrated circuits to maximize performance, while guaranteeing reliable operation under process variations. Each stage is described in detail in the following sections.

3.1 Topology Generator

In the topology generator, we represent each circuit topology as a graph and employ a graph generation algorithm to obtain candidate circuit structures. The graph-based method in [4] gives an example of expressing digital circuits as a graph, where pull-up and pull-down networks are generated separately, and each transistor corresponds to an edge in the graph. The two nodes connected by an edge define the source and drain, whereas the gate connection is defined as one of the node properties. However, this approach is not applicable to other types of circuits that do not have separate pull-up and pull-down paths, where N-channel and P-channel MOSFET devices can be placed more arbitrarily. Therefore, we propose a generalized graph representation method suitable for a broader range of circuits shown in Fig. 3.2. In our representation method, an edge (transistor) has *gate* and *size* properties, representing the net connected to the gate and transistor size. A Node has a *type* property that represents the net type (e.g., input port, output port, supply, ground, and internal net). Additionally, we introduce a new property *voltage* in the nodes. This property represents a relative voltage of each node and has a range of [-1, 1]. The *voltage* of an edge is obtained by

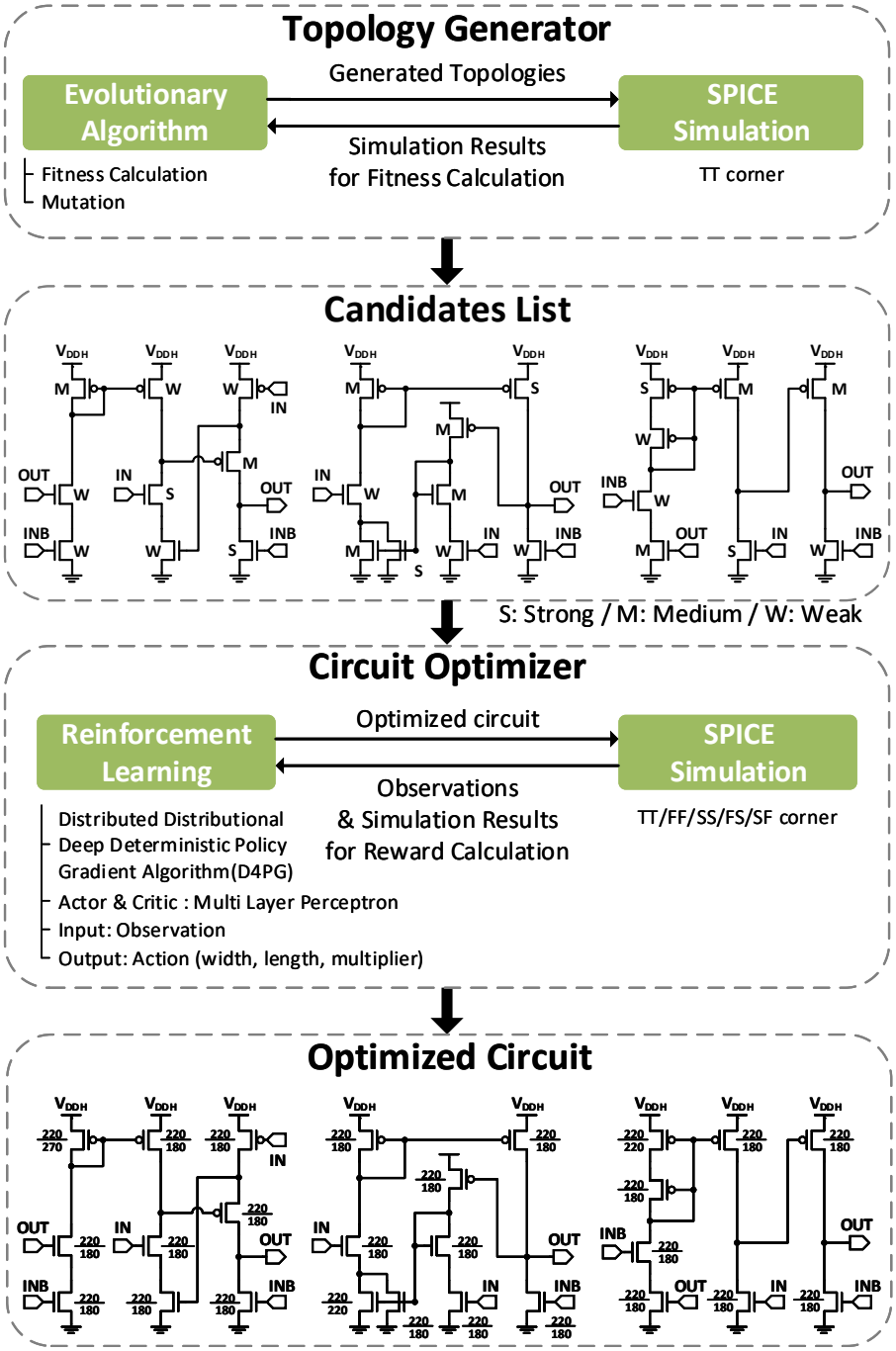
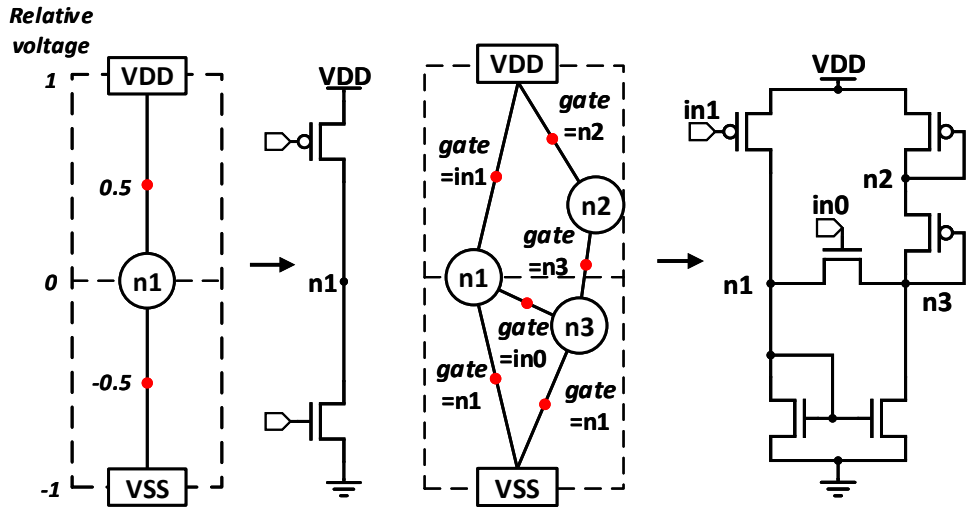


Fig. 3.1: Overview of the proposed circuit design framework.



(a) Simple circuit with a pair of MOSFET devices

(b) Complex circuit where gates are connected to other nodes

Fig. 3.2: Examples of proposed graph-based circuit representation.

averaging the voltages of the nodes on both ends (source and drain). The edges with positive *voltage* translate to P-channel MOSFET devices, whereas the edges having zero or negative *voltage* represent N-channel MOSFET devices. This method allows for generating more generalized circuit structures while preserving a common circuit property that P-channel MOSFET devices are typically placed near the power supply voltage, whereas N-channel MOSFET devices are biased at lower voltages to maximize operation range.

Since we aim to generate an optimal circuit topology without prior knowledge, we suggest employing an genetic algorithm in the topology generator. NeuroEvolution of Augmenting Topologies (NEAT) is a widely used genetic algorithm for exploring artificial neural network structures [25]. The algorithm starts from a simple network with a single fully-connected layer, and the network evolves into more complex structures through crossover and mutation over generations. We modify the NEAT algorithm to make it suitable for circuit topology generation; we introduce the voltage concept into

Algorithm 1 Topology Generator

Input: Population size N, Max Generations G, Mutation Probability

Output: Candidate Topologies

P: Population, C: Offspring, P_0: Initial Population
for $g = 1, 2, ..., G$ **do**
  Simulate all $C_i (i \in N)$ and Calculate Fitness
  Remove Stagnated Species and Extract Candidates
  Calculate Fitness of Species $s_k \in S_g$
  Calculate Reproduction Size R_k of each s_k
  for all s_k in S_g **do**
    Add Best Candidate in s_k to P_{g+1}
    Make Parent Pool with N' Top Candidates in s_k
    for $j = 1, 2, ..., R_k$ **do**
      Crossover
      Mutate
      Add C_j to P_{g+1}
    end for
  end for
  Speciate P_{g+1}
end for

the NEAT algorithm, and the mutation functions and the properties of genes are heavily modified aimed at circuit topology generation.

Algorithm 1 details the proposed circuit topology generation algorithm. An offspring represents a candidate circuit topology and has node and connection genes. The node gene represents a node in the graph and has *type*, *voltage* and *innovation number* properties. The *type* property determines the type of the nodes (input port, output port, supply, ground, or internal net) and the *voltage* property represents its relative voltage, whereas the *innovation number* is a unique identifier. The connection genes represent

edges in the graph with *in*, *out*, *size*, *gate*, and *innovation number* properties. The *in* and *out* properties define two endpoints of the edge (source and drain of the transistor), and the *size* property defines the relative strength of the transistor. As described above, since a transistor is a three-terminal device, the node to which the gate of the transistor is connected is defined by the *gate* property. The *innovation number* is a unique identifier. A population is a set of all offspring of the current generation. The population is divided into several species based on similarity. Each species has a base model, and the offspring close to the base model are included in the species.

The topology generator first creates an initial population P_0 which consists of offspring with only three node genes and two connection genes: V_{DD}, V_{SS}, and an internal node connected by one P-channel MOSFET & N-channel MOSFET pair (Fig. 3.2(a)). The gate of each transistor is randomly connected to the node except V_{DD} and V_{SS}. In each generation, evolution begins by calculating the fitness of the species in the current population. The algorithm converts each offspring into a netlist and runs a SPICE simulation to calculate the offspring's fitness based on the observed functionality and performance. Then, the fitness of the offsprings included in the species is averaged to obtain the species fitness. The number of offspring that can reproduce from each species to the next generation is determined in proportion to the species fitness. During circuit topology generation, simulations are performed only at the typical (TT) corner to scan large search spaces and find the best candidates quickly.

Before reproducing a new population, the algorithm observes whether the fitness of the best offspring in each species has been improved or not in the last few generations. If the fitness of a species does not improve any further for a certain number of generations, then the species is considered stagnant, and offsprings of that species are removed from the population. The evolution process is independently performed for each species. First, the offspring with the highest fitness in each species is automatically included in the population of the next generation. Next, a set of offspring with the highest fitness within each species is selected as a parent pool. Two offspring

are randomly chosen from the pool and compared with each other, where the winner evolves through mutation and joins the population of the next generation.

The fitness function represents the performance and reliability of a circuit as a single value. We consider two types of design constraints for fitness calculation: hard constraints and soft constraints. The hard constraints are the set of design constraints that a circuit must satisfy (e.g., rail-to-rail output swing for level shifters), whereas the soft constraints indicate the design quality (e.g., power consumption and conversion delay of level shifters). The fitness of an offspring at the x-th generation is calculated as

$$fit_x = \sum_{i \in H} \alpha_i f(q_{i,x}) + \{\prod_{i \in H} f(q_{i,x})\}\{\sum_{j \in S} \alpha_j f(q_{j,x})\} \tag{3.1}$$

where fit_x is the calculated fitness of an offspring, $q_{i,x}$ is the observed performance of the circuit in SPICE simulations corresponding to the i-th constraint, $f(q_{i,x})$ is the score function for each constraint, α_i is the weight of the i-th constraint, and H and S represent the sets of hard and soft constraints, respectively. This is similar to the reward function used in RL for circuit optimization in [19], but our approach has two distinct differences: i) we use $\log(q_{i,x})$ instead of $q_{i,x}$ for the scores that have a large dynamic range, and ii) the contribution of soft constraints in the fitness is regulated by the scores related to the hard constraints, instead of using a hyper-parameter manually tuned for a specific type of circuit. In early generations, it is highly likely that most offspring would fail to function properly. The scores related to the hard constraints would be very low, making the fitness largely dictated by the hard constraints. Hence, the algorithm focuses on finding feasible topologies that produce a desired output. Once the algorithm finds properly working circuit topologies, the scores related to the hard constraints saturate and do not affect the fitness. The remainder of the evolution process further modifies the topology to improve circuit performance.

The topology generator employs various mutation functions so that it can cover a wide range of circuit topologies. Note that the nodes without any connection (i.e., floating nodes) can be generated as a result of mutation. Hence, we label the nodes

with one or more connections as active nodes, and only active nodes are selected for mutation. The types of mutations are discussed below:

Add connection **:** This mutation randomly chooses two active nodes and connects them by adding a new edge. Since an edge corresponds to a transistor in the actual circuit, it links the gate of the new edge to one of the existing active nodes by updating the *gate* property.

Add node **:** This inserts a new node in one of the edges. In other words, a single transistor is replaced with two stacked transistors. The gates of the stacked transistors are connected to the same node to which the gate of the original transistor was connected. This process is often used when designing a circuit to increase output resistance or minimize leakage current.

Add P-channel MOSFET & N-channel MOSFET pair **:** A P-channel MOSFET & N-channel MOSFET pair makes a new connection between V_{DD} and V_{SS}. If a single P-channel or N-channel MOSFET transistor is placed between V_{DD} and V_{SS}, this will be just a current leaking path. Therefore, we place transistors as a pair of P-channel and N-channel MOSFET devices when making a new connection between the supply rails.

Change gate **:** The gate of a transistor is connected to a different active node except for V_{DD} and V_{SS} nodes.

Remove connection **:** This mutation randomly removes one of the connections, which allows for removing transistors from the current topology. This prevents the circuit from continuously becoming larger.

Change size **:** The size of the connection genes represents the relative size (strength) of a transistor. Since our goal is to quickly go through a variety of circuit topologies and find promising candidates, we define each transistor's strength in only three steps: strong, medium, and weak. During mutation, transistor size randomly changes in each connection gene independently.

Change output port **:** This mutation changes the location of the output port. One

of the active nodes is selected as an output.

In the original NEAT algorithm, each mutation function is randomly selected in each mutation. Hence, multiple types of mutations may be performed simultaneously. However, this may result in an excessive amount of change in a circuit. For instance, removing a transistor from the circuit and changing the gate connection of another transistor would produce a circuit with entirely different characteristics. Hence, we limit the mutation process to select only one of the add, remove or gate change mutations (mutations 1 through 5 above). In addition, other minor mutations (mutations 6 and 7) are independently introduced with a certain probability. Let $P_{addNode}$, P_{addCon}, $P_{addPair}$, $P_{changeGate}$, and P_{rmCon} denote the probability of mutations 1 through 5 above. Then, the mutation process follows the equation below:

$$P_{addNode} + P_{addCon} + P_{addPair} + P_{changeGate} + P_{rmCon} = 1 \tag{3.2}$$

During topology exploration, we do not want the algorithm to keep adding transistors indefinitely. Otherwise, the number of transistors in a circuit may explode, and the resulting circuit would be far from what we desire. For instance, an ideal analog amplifier or level shifter circuit typically has tens of transistors at most. Therefore, we balance the expected number of removed and added transistors in each mutation by enforcing the relationship below:

$$2P_{addNode} + P_{addCon} + 2P_{addPair} - P_{rmCon} = 0 \tag{3.3}$$

since adding a node (a net in the circuit) adds two transistors, whereas adding or removing a connection adds or removes a single transistor in the circuit.

After a new generation is obtained by mutating all the offspring of the current generation, the newly generated offspring are grouped again into a set of species. Each offspring is compared to the base models of existing species. If the number of differences in the connection genes is below the threshold for one or more existing species, then the offspring joins the closest species. Otherwise, the offspring constitutes a new species and becomes its base model. After the grouping process is done, the population

undergoes another iteration of the mutation process to obtain the next generation. This process continues until it reaches the maximum number of generations defined by the user.

The topology generator selects candidate topologies both during and at the end of the evolution. When a stagnant species is removed during evolution, the offspring with the best fitness in that species is selected and added to the candidate list if it meets all the given design constraints. When the algorithm finishes the last iteration, the same operation is performed on all the remaining species. Note that there may exist floating nodes and floating paths as a result of mutation. Before adding an offspring to the candidate list, the topology generator finds and removes the floating nodes and paths.

3.2 Circuit Optimizer

The topology generator is aimed at quickly finding promising circuit topologies. Hence, each transistor is only roughly sized during exploration (e.g., strong, medium, or weak). This accelerates the search process by significantly limiting the search space, but the size of each transistor must be further tuned for optimal performance. For this purpose, we employ an additional circuit optimizer as the second stage in the proposed circuit design framework.

The circuit optimizer adopts a reinforcement learning (RL) algorithm to optimize candidate circuits. Various RL algorithms have been used for circuit optimization. L2DC [19] and GCN-RL [21] are based on deep deterministic policy gradient (DDPG) [31], and AutoCkt [20] adopts proximal policy optimization (PPO) [35]. DDPG has an actor-critic structure and generally works well in continuous or high-dimensional action spaces. An agent collects and saves a sample into a replay memory. Then, a mini-batch is randomly selected from the replay memory to train the network. While PPO also has an actor-critic structure suitable for training in continuous or high-dimensional action space, it does not have a replay memory. Instead, N agents collect

samples in parallel during an episode which consists of T time steps, and a mini-batch is constructed using the collected samples and used for training the algorithm. Then, all the samples are discarded. DDPG exhibits slower convergence during training since it only uses one agent contrary to PPO, but has the advantage of being able to reuse the samples stored in the replay memory. PPO trains the model more quickly by using multiple agents, but it only uses the samples collected in the current episode for training, which reduces sample efficiency. In circuit optimization, samples are obtained by running time-consuming SPICE simulations. Therefore, it is crucial to maximize sample efficiency (i.e., reduce the number of samples required for algorithm convergence) to speed up the circuit optimization process. To resolve this issue, we adopt distributed distributional deep deterministic policy gradient (D4PG) [34] algorithm in the circuit optimizer. D4PG supports both multi-agent training and sample reuse by using a replay memory. Unlike DDPG and PPO which express future rewards as a single scalar value, D4PG expresses rewards as a probability distribution. It models the inherent uncertainty imposed by function approximation in a continuous environment, resulting in better gradients and improving the training performance compared to DDPG. It also shows more stable performance when multiple agents are used [34].

In the RL algorithms using actor-critic structure, two different neural networks are typically employed: an actor network and a critic network. The actor network takes a state vector as an input and produces an action vector, whereas the critic network takes state and action vectors as inputs and predicts the reward value an agent is expected to receive as a result of the current and future actions. The RL algorithm trains those neural networks on the observed samples. As the complexity of the neural network increases with the dimension of input vectors, it is important to minimize the dimension of the input vector for faster optimization. Since the action vector represents relative size changes of all the transistors in the circuit, its dimension is fixed. Hence, we aim to optimize the critic network by reducing the dimension of the state vector. Specifically, we use the simulated circuit performance (e.g., power consumption and delay) and

area as a state, instead of feeding each transistor's size or other characteristics (e.g., V_{th}, V_{sat}, and μ_0) as did in prior works [19–21]. Therefore, the dimension of the state vector is independent of the number of transistors in the topology and the optimization process can be efficiently accelerated when the target circuit topology consists of many transistors.

The actor network creates an action based on the state obtained by SPICE simulations. An action represents a relative change in the size (width, length, and multiplier) of each transistor. If the target topology has *N* transistors in total, the dimension of the action vector would be 3*N*. Each dimension of the action vector has a value in [-1, 1]. Then, the amount of change in the size of the i-th transistor is

$$\Delta Size = round(Action \cdot \frac{size_{max} - size_{min}}{L_{maxStep}}) \tag{3.4}$$

where $\Delta Size$ is the amount of change in transistor size, $Action$ is the output of the actor network, $L_{maxStep}$ is the number of steps in one episode, and $size_{max}$ and $size_{min}$ are the allowable maximum and minimum transistor sizes. This translates to the maximum size change that can occur in one episode equal to $size_{max} - size_{min}$. The size values are real numbers, so they are rounded to the closest values allowed in the given process before converted to an actual circuit.

In D4PG, when a sample collected by one of the agents is stored in the replay memory, a mini-batch is created by randomly choosing samples from the replay memory. However, as the training progresses, the amount of samples stored in the memory becomes larger; thus, only a fraction of stored samples is used to generate a mini-batch, reducing sample efficiency. In addition, the learner updates the network only once in each time step, and the SPICE simulation to obtain a new sample becomes the processing bottleneck. To address these issues, we propose to adopt a multi-update technique that has been used for unbiased learning. When a sample is obtained by the agent and stored in the replay memory, unlike the conventional method of circuit optimization that creates one mini-batch from the stored samples, we create several mini-batches and update the critic and actor networks multiple times. This accelerates the circuit op-

timization process without time overheads since multiple updates could be performed while SPICE simulations are running. This scheme also allows for unbiased learning through random sampling that removes correlation between mini-batches, reducing the possibility of overfitting.

At the beginning of training, the actor network tends to generate the same action even if the state changes gradually in each step. In other words, the size of a transistor continues to increase or decrease regardless of the current state. This is because the output is close to either 1 or -1 in most cases when the actor network weights are randomly initialized. The actor network typically uses the tanh function as the activation function. In a randomly initialized network, the output of the network, which is the input to the final tanh activation function, typically has an absolute value of 2 or larger, rendering the final output close to ± 1. This effect is amplified by the fact that circuit performance is converted to a state using a logarithmic function. Even if the state changes, the sign of the action which determines size change direction (increase or decrease) is likely to stay the same. In addition, the weights of the actor network in each agent are updated only when an episode ends, and they remain fixed for all the steps within an episode. Therefore, in the first few episodes, the sizes of many transistors just move to the minimum or maximum value. This severely hinders circuit optimization by moving the design far from the initial point, which is already a near-optimal design found in the circuit topology generator. To solve this problem, we propose an episode early stopping technique that limits the number of steps in an episode in the early stage of training. As the learning progresses, it gradually increases the number of steps in each episode, and the episode finally proceeds with the maximum number of steps defined by a hyperparameter. This technique allows the network to learn more stably while acquiring more meaningful samples near the initial point in early episodes.

Algorithm 2 details the proposed circuit optimizer. The exploration agent gets an action by entering the current state into the actor network in each step. The algorithm

Algorithm 2 Circuit Optimizer

Learner

Input: Number of Steps in Episode N, Batch Size M, Replay Memory Size R, Learning Rates α_0 and β_0, Multi-Update Parameter U

Determine Network Size by Analyzing Netlist
Initialize Network Weights with Kaiming Initialization
for $i = 1, 2, ..., N$ **do**
    Wait for Samples from Agents
    for $j = 1, 2, ..., U$ **do**
        Randomly Choose M Samples from Replay Memory
        Compute Updates of Actor and Critic Networks Using Samples
        Update Network Parameters
    end for
end for

Agent

Input: Number of Steps in Episode N, Number of Actors P, Episode Early Stopping Interval T

repeat
    Initialize Episode
    Copy Actor Network from Learner
    for $step = 0, ..., K$ **do**
        Get Action from Actor Network and Change Size
        Simulate and Calculate State (s) and Reward (r)
        Send Sample to Learner
    end for
    Increase *K* every *T* Episodes
until Learner Finishes

uses the network output (action) to change the size of transistors with the corresponding action values and runs a SPICE simulation to obtain the reward and the state. The reward function in the circuit optimizer is identical to the fitness function (Eq. 3.1) employed in the topology generator, except that the scores are obtained at different process corners, as explained later in this section. The current state, the action, the next state, and the calculated reward constitute a single sample and are written to the replay memory. Each time a new sample is sent to the replay memory, the optimizer creates multiple mini-batches to update the neural networks. This update process continues until it reaches a user-defined maximum number of steps. Then, the best set of the parameters found in the course of training is selected as the final design.

L2DC [19] uses a Recursive Neural Network (RNN) in the actor network and Multi-Layer Perceptron (MLP) as the critic network. However, RNN is typically hard to train due to the vanishing and exploding gradient problems [38]. Also, the state is composed of the observed values (e.g., g_m and $\mathrm{V_{th}}$) of each transistor, and the order is determined by the signal path of the circuit, necessitating manual examination of the circuit topology. Instead, we use an MLP as the actor as did in AutoCkt [20], where the specifications of topology are combined into a state vector in an arbitrary order. Also, we initialize the weights of the MLP following the method in [39].

While the circuit optimizer primarily focuses on maximizing circuit performance, it is also very important to guarantee that the circuit properly operates under process variations. Contrary to prior works on circuit optimization [19–21], we run SPICE simulations at five different process corners (TT, FF, SS, FS, and SF). The optimizer constantly observes if the circuit meets the hard constraints at all corners during optimization. Contrarily, the scores related to the soft constraints are only measured at the typical (TT) corner. This allows the circuit to exhibit maximum performance at the corner of most concern while still guaranteeing proper functionality in the worst cases. Note that Monte-Carlo analysis better captures the robustness of a circuit under process variation. However, since the size of each transistor continues to change dur-

ing optimization, adopting Monte-Carlo analysis will require a large number of SPICE simulations in each time step, incurring a large time overhead. Contrarily, the corner analysis requires only a few simulations for each design point and hence is more suitable for fast optimization.

Chapter 4. Experiment Result

In the previous section, we presented an unified circuit design framework that automatically generates appropriate circuit topologies and further optimizes each design through finding an optimal size of each transistor. In this section, we experimentally verify the proposed circuit design framework. By employing the framework to design level shifter circuits, we demonstrate that the topology generator produces novel level shifter topologies, and the circuit optimizer successfully improves the design. Finally, the resulting level shifter designs are fabricated and compared against prior arts designed by human experts. All experiments are conducted on a workstation running CentOS 7.4 with two Intel E5-2687W v4 processors, 128GB DRAM, and an Nvidia GTX Titan X GPU. The topology generator only uses the processors whereas the circuit optimizer uses both the processors and GPU.

4.1 Level Shifter Design

We choose a level shifter circuit as a test vehicle for our framework since it is an active research area where new circuit topologies are continuously developed. There are many different topologies, and an optimal topology varies with the design constraints [24]. Therefore, the effectiveness of our framework that is capable of finding optimal circuit topologies could be verified more clearly. In addition, level shifter circuits share common properties both with digital and analog circuits. For instance, level shifters operate on a rail-to-rail input signal and produce a rail-to-rail output in a higher voltage, similar to digital circuits [22]. On the other hand, the internal operation is similar to that of analog circuits such as amplifiers. In experiments, we adopt the framework to design level shifter circuits in a 180nm CMOS process, and the resulting circuits are compared to prior designs reported in the literature.

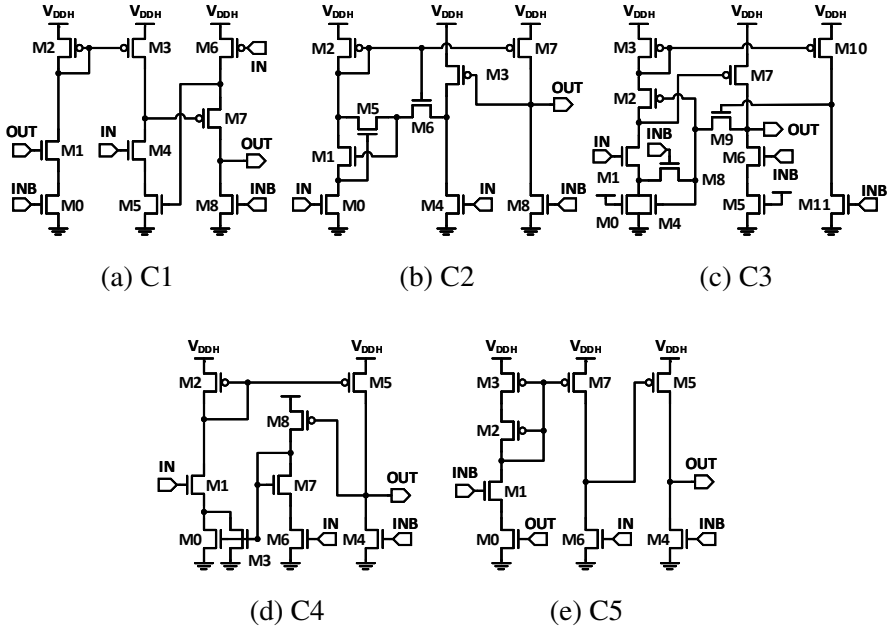
A level shifter circuit converts a low-voltage (V_{DDL}) digital signal to a high-voltage (V_{DDH}) signal. Level shifters must generate a rail-to-rail swing between the ground and V_{DDH} at the output. Therefore, we use output signal swing as a hard constraint in the framework. Because level shifters are typically expected to operate with high conversion speed and low power consumption with minimal footprint [40], we use delay, total power (P_{total}), static power (P_{static}), and area as soft constraints. The circuit area is calculated as the number of transistors in the topology generator, whereas the circuit optimizer uses the total active area.

Table 4.1: Experimental setup for level shifter design

		Topology Generator	Circuit Optimizer
Generation & Step		400	350,000
Process Corners		TT	TT/FF/SS/FS/SF
Hard Constraint	Swing Ratio	$q_{i,x}$ / V_{DDH}	$q_{i,x}$ / V_{DDH}
Soft Constraints	Delay	$-logq_{i,x} + b$	$-logq_{i,x} + b$
	P_{total}	$-logq_{i,x} + b$	$-logq_{i,x} + b$
	P_{static}	$-logq_{i,x} + b$	$-logq_{i,x} + b$
	Area	$1 - \frac{max(q_{i,x}-b,0)}{slope}$	$-logq_{i,x} + b$

Table 4.1 shows the score functions used in each step. The scores related to the soft constraints are calculated as -log(qi,x) except for the area in the topology generator, whereas the score for the hard constraint (output swing) is calculated as the swing observed in simulation divided by V_{DDH}. In topology generation, the area is calculated as the number of transistors in the circuit. When the number of transistors exceeds a threshold (b in Table 4.1), the score is divided by a slope which is a hyperparamter. In the circuit optimizer, we use the worst values across all process corners when calculating the score for the hard constraint. Soft constraint scores are obtained at the TT

corner.



(a) C1 (b) C2 (c) C3

(d) C4 (e) C5

Fig. 4.1: Level shifter circuit topologies generated by topology generator

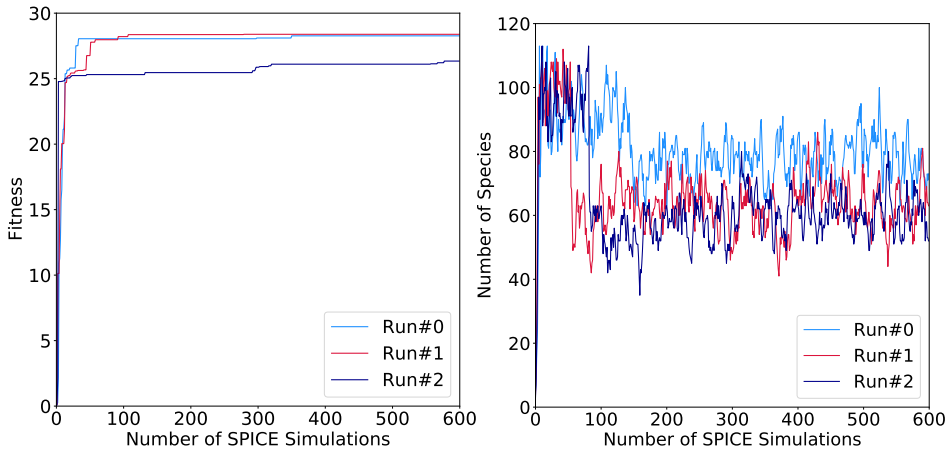
4.2 Topology Generation

The topology generator runs seven SPICE simulations in parallel only at the TT corner for fast topology search. The input inverter of level-shifter is implemented using low threshold voltage (V_{th}) devices, whereas the other transistors are standard V_{th} devices. We use a minimum-sized transistor with 180nm channel length and 220nm channel width as a weak device. Medium and strong devices have 2× and 4× larger channel width, respectively. The initial population size is set to 450, and the population evolves for 600 generations, which takes approximately 5h 30m. In addition, we experiment with varying the soft constraint weights in the fitness function to observe how the topology generator performs under different design constraints. The generated circuit topologies are displayed in Fig. 4.1. Specifically, three cases are tested: i) all the

Table 4.2: Results of topology generation

V_{DDL}=0.4V @ 1MHz						
	Swing Ratio	P_{total}(nW)	P_{static}(nW)	Delay(ns)	Transistors	Fitness
C1	1.00	47.3	0.10	22.2	9	28.39
C2	0.98	52.2	0.22	11.9	9	26.43
C3	1.00	50.5	0.70	17.8	12	26.36
C4	0.99	58.0	0.71	16.6	9	27.38
C5	1.00	39.8	0.65	15.0	8	27.75

constraints have the same weight (C1 in Fig. 4.1), ii) only the weight of static power is lowered (C2-C3), and iii) the weight of delay is increased while the weights of static and total powers are decreased (C4-C5). Table 4.2 summarizes the performance and fitness of the generated circuits (C1-C5). Simulation results show that the circuits generated with lower static power weight (C2-C3) exhibit higher static power than C1. In addition, the circuits optimized for delay (C4-C5) achieve lower delay than C1 at the expense of power consumption increases.



(a) Fitness (b) Number of species

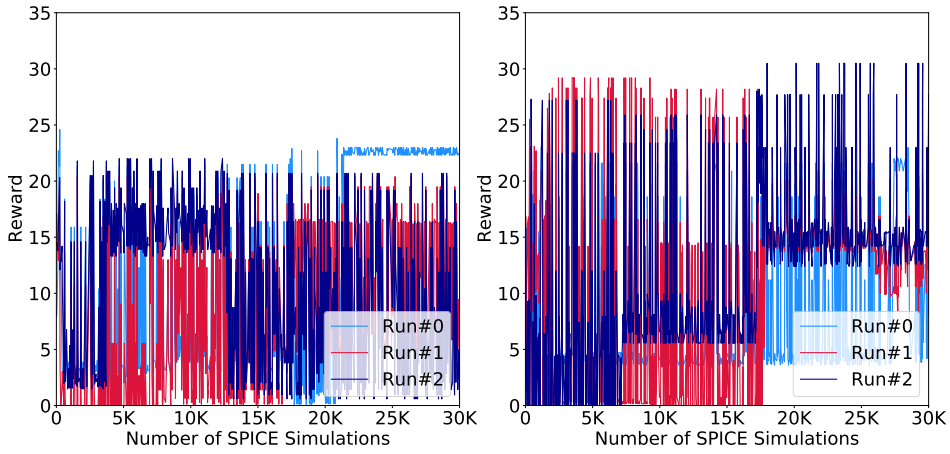
Fig. 4.2: Experimental results of topology generation.

We also perform three independent runs of circuit topology generation to estimate algorithm stability. Fig. 4.2 shows the fitness and the number of species as the evolution proceeds. The best fitness, which is the fitness of the best circuit in the population, rapidly increases in the first 7-9 generations, and then gradually improves through fine tuning of the circuit topology. Note that the value of fitness is not capped at a fixed value. While the fitness of a circuit can have an arbitrary value, the generated circuits exhibit fitness values less than 30 in our experiments. The number of species is nearly constant during evolution except in the first few generations, suggesting that stagnant species are replaced with a similar number of new species.

4.3 Circuit Optimization

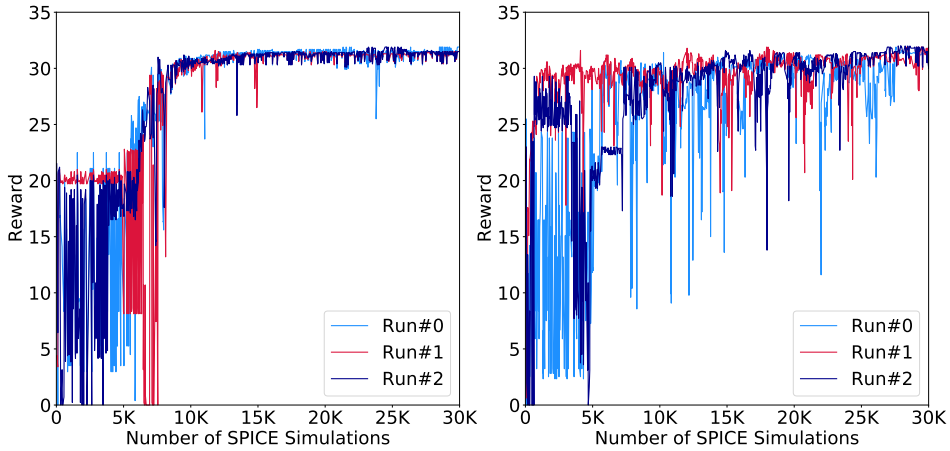
In experiments, we use an MLP with three hidden layers and 200 nodes in each layer as the actor network. The critic network has the same structure but has two hidden layers. First, we evaluate each of the proposed RL optimization techniques using WCMLS circuit, which is widely adopted in level shifter designs [22–24]. Experiments are performed using a total of seven actors, where one of them is used to estimate the performance of the optimization algorithm in real time (evaluation actor). 30,000 SPICE simulations are run across all the actors except the evaluation actor, which takes 2h 20m. Since the RL algorithm has some degree of randomness, we test each configuration on three independent runs to observe its reliability. Fig. 4.3-4.5 summarizes the experimental results. Fig. 4.3(a) shows that conventional D4PG fails to converge in two of the three runs. However, when the multi-update technique with U=10 is applied, the algorithm successfully finds a correct optimization direction and properly biases transistors in the circuit after about 7,000 SPICE simulations (Fig. 4.4(a)). Fig. 4.5(a) shows the optimization results when the episode early stopping method is also employed. Initially, an episode stops only after four steps, and the episode length increases by two after every five episodes in each exploration agent until it reaches the

maximum length of 20. This method reduces the number of SPICE simulations required to capture the bias points from 7,000 to 5,000, suggesting that this technique accelerates RL training in the early stage. Note that the algorithm shows more fluctuation during optimization when the early stopping method is adopted. We suspect that the conventional approach is exposed to more ”bad” samples, which are far from the initial nearly-optimized design from the topology generator, in the early stages of training. Those samples exhibit very low rewards as they do not meet the hard constraints. As a result, the actor is trained to be more conservative, and once the design enters the near-optimal region where the hard constraints are satisfied, the algorithm tends to stay near that point only with fine tuning to avoid a large drop in the reward value. Contrarily, the episode early stopping method allows the design to enter the near-optimal region quickly, significantly reducing the number of bad samples during initial training. When the design approaches an optimal point during optimization, the algorithm now searches for better design points more aggressively. In other words, the algorithm is less reluctant to depart from the local optima, which helps find a global optimum.



(a) D4PG (b) DDPG

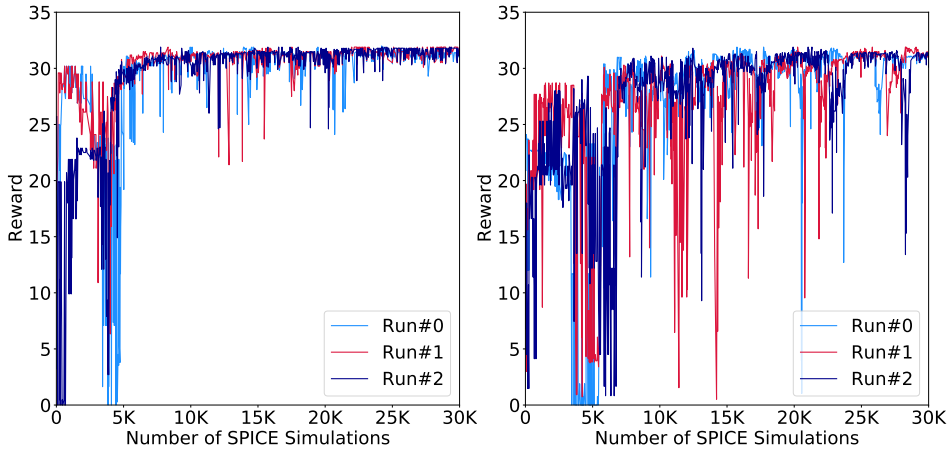
Fig. 4.3: Trends of reward improvement without techniques.



(a) D4PG with multi-update (b) DDPG with multi-update

Fig. 4.4: Trends of reward improvement with multi-update techniques.

For comparison, we experiment with the DDPG algorithm adopted in prior work [19] using the same environment. D4PG, which is employed in our framework, is similar to DDPG except that it uses multiple agents in parallel, and the output of the critic network is represented as the probability distribution. The DDPG algorithm is trained for 30,000 SPICE simulations in total, and the total running time is 14h 30m. This is more than six times longer than the time required for our approach to process the same number of SPICE simulations, which confirms the effectiveness of the multi-agent training of D4PG. The experimental results are displayed in Fig. 4.3-4.5. We experimented with a vanilla DDPG algorithm (Fig. 4.3(b)), DDPG with multi-update (Fig. 4.4(b)), and DDPG with both techniques (Fig. 4.5(b)). Experimental results show that DDPG exhibits larger variations between runs and unstable training convergence compared to our approach. To observe how the type of critic affects the training performance, we experimentally apply the scalar value critic used in DDPG to our algorithm. With both multi-update and episode early stopping applied, Fig. 4.6(a) shows that using the scalar value critic results in more unstable training convergence compared to Fig. 4.5(a).

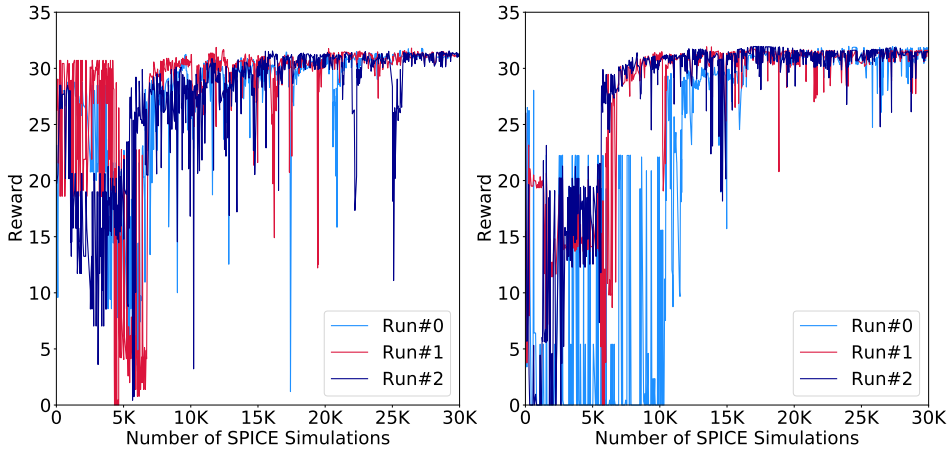


(a) D4PG with multi-update and episode early stopping

(b) DDPG with multi-update and episode early stopping

Fig. 4.5: Trends of reward improvement with multi-update and episode early stopping.

The episode early stopping method effectively limits the agent's exploration capability in the early stages of training, and a similar effect could be achieved by scaling the output of the actor network. We conducted additional experiments in which we multiplied the output of the actor network with a scaling factor before passing it to the environment. The scaling factor is set to 0.2 at first and is increased by 0.1 every five epochs, which translates to the maximum amount of size change in each episode identical to the episode early stopping method. Experimental results are displayed in Fig. 4.6(b). It can be seen that this scaling method results in a slower convergence. We suspect that this is because the actor network is not properly trained in early episodes due to the continuously changing scaling factor. More specifically, the actor network is trained in a way to generate the best action for the current state. However, the output of the actor network is scaled before being applied to the environment, and hence the actor should take this into account during training. Since we are now changing the scaling factor, the actor should be trained in different directions as the optimization process continues, hindering proper training.

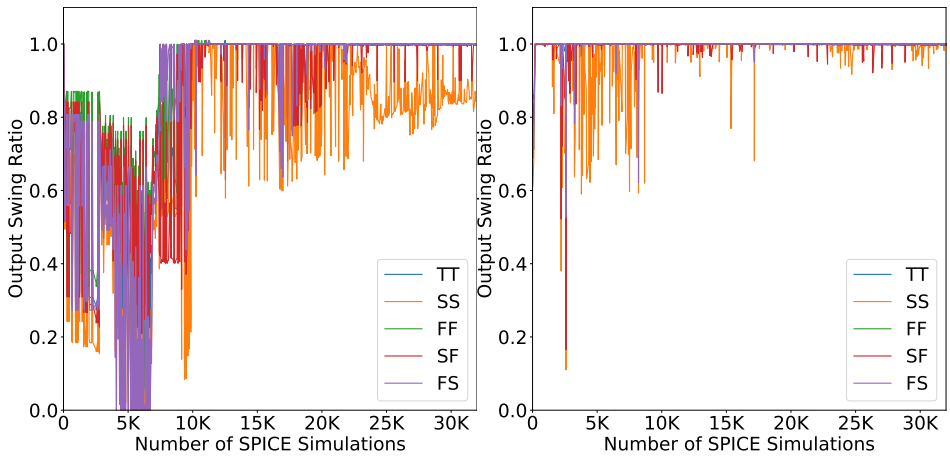


(a) Scalar value critic (b) Action scaling

Fig. 4.6: Reward trends of alternative approaches for comparisons.

During optimization, our framework considers multiple process corners to make sure the circuit properly works under process variations. Fig. 4.7 compares our approach to the conventional method that observes the circuit performance at the TT corner only. When the circuit is optimized only at the TT corner, the relative output voltage swing reaches 0.95 at the same corner, but the design may produce much smaller swing at different corners (Fig. 4.7(a)). On the other hand, if we obtain the score related to the hard constraint at the worst corner during optimization, the resulting circuit achieves $>$0.95 output swing at all the corners.

Similar to topology generator, we also experiment with changing the weights of the soft constraints. The sum of the weights is fixed, and their values are allocated differently in each case. Table 4.3 summarizes experimental results for optimization with 32,000 SPICE simulations. The last column shows the actual weights of the soft constraint of interest and the others. As expected, increasing the weight for total power consumption further reduces power consumption during optimization while sacrificing delay and area since the circuit is subject to a trade-off between delay, power, and area. Similarly, using a higher weight for delay produces a faster level shifter circuit at the



(a) Optimization at TT corner only (b) Optimization at all corners

Fig. 4.7: Trends of output swing ratio with TT only and all corner. Considering process corners significantly improves reliability.

Table 4.3: Experiments with different weights in circuit optimization

V_{DDL}=0.4V @ 1MHz						
Prioritized Metric	Swing Ratio	P_{total} (nW)	P_{static} (nW)	Delay (ns)	Area (μm^2)	Weight*
None	0.97	27.5	1.05	22.5	1.06	1.0/1.0
P_{total}	0.98	26.2	1.05	22.3	1.07	1.6/0.8
	0.99	25.1	0.93	29.6	1.26	2.3/0.6
	0.97	25.0	0.93	27.9	1.18	2.9/0.4
	0.96	24.7	0.89	31.9	1.24	3.1/0.3
Delay	0.97	26.6	1.09	21.1	1.00	1.6/0.8
	0.99	26.6	1.10	20.8	1.03	2.3/0.6
	1.00	27.5	1.16	20.7	1.20	2.9/0.4
	1.00	62.6	6.41	17.9	4.32	3.1/0.3

* Weight of prioritized metric / weight of other metrics

Table 4.4: Results of optimizing generated circuits

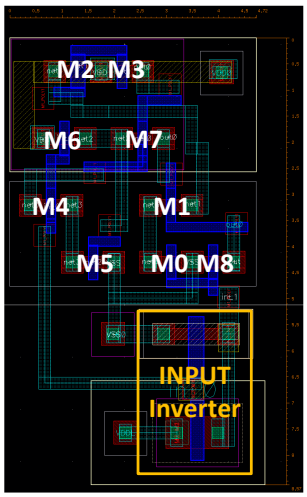
V_{DDL}=0.4V @ 1MHz						
	Swing Ratio	P_{total}(nW)	P_{static}(nW)	Delay(ns)	Area(μm^2)	RW
C1	1.00	26.2	0.30	11.4	0.38	33.35
C2	1.00	34.2	0.16	9.7	0.45	33.49
C3	1.00	34.3	0.36	12.5	0.68	33.02
C4	0.99	37.3	0.32	9.8	0.37	33.28
C5	1.00	26.0	0.43	11.3	0.32	33.25

expense of power and area increase.

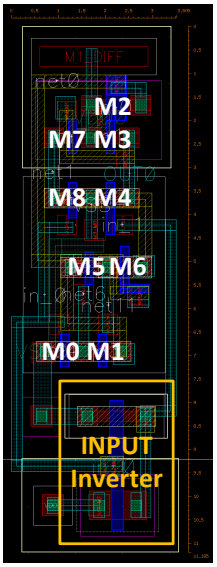
Finally, we apply our circuit optimizer to the circuits generated by the topology generator (C1-C5). Similar to previous experiments, we use seven actors in the RL algorithm, where one of them is used as an evaluation actor. For each circuit topology, 38,000 SPICE simulations were performed except the evaluation actor, and the multi-update constant U was set to 13. The optimizer successfully improved all the generated circuit topologies, which is verified by comparing the results in Table 4.4 to the results in Table 4.2. Note that the area represents the total active area, not the actual layout size.

4.4 Test Chip Fabrication

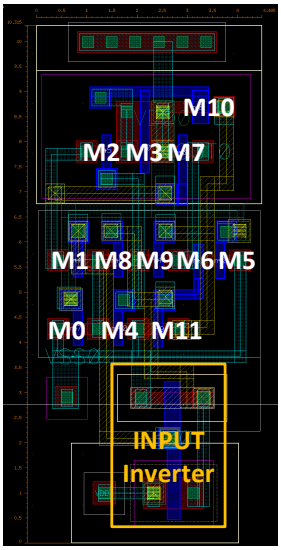
To validate level shifter circuits designed by our framework, we fabricated the generated and optimized circuits C1-C5 in a 180nm process. Since the framework only provides a netlist as the output, the layout was manually drawn, as shown in Fig. 4.8. The input inverter supplied by V_{DDL} is included in the layout. It is difficult to measure the conversion delay of a level shifter accurately, since parasitic components (e.g., I/O cell, PCB trace, and bond wire) also contribute to the delay. Hence, we adopt the dual-path measurement method in [41]. Two different paths with and without a level shifter



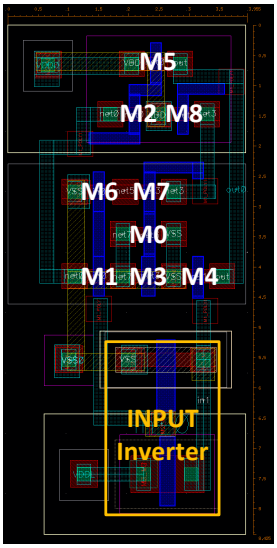
(a) C1 (4.7µm×8.6µm)



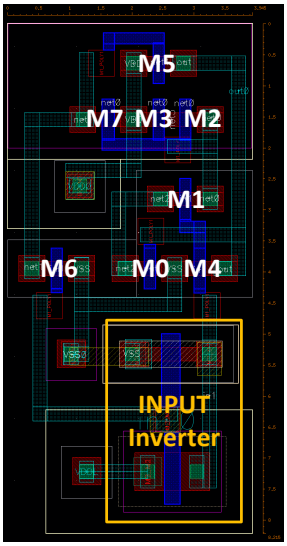
(b) C2 (3.5µm×11.2µm)



(c) C3 (4.5µm×10.3µm)



(d) C4 (4.0µm×8.4µm)



(e) C5 (4.0µm×8.2µm)

Fig. 4.8: Layout of generated circuits and their size.

are implemented, and the conversion delay is indirectly measured by subtracting their delays as depicted in Fig. 4.9. The V_{DDL} inverter (colored gray in the figure) converts a high-voltage input to a low-voltage signal, which is later converted back to V_{DDH} by the level shifter. Each level shifter has a dedicated power supply rail to measure its power consumption. A different level shifter can be selected by a control signal to the multiplexer and demultiplexer. The conversion delay is measured as the difference in arrival times of OUT and REF signals. Fig. 4.10(a) shows the top-level layout of the test chip, and Fig. 4.10(b) is the chip micrography.

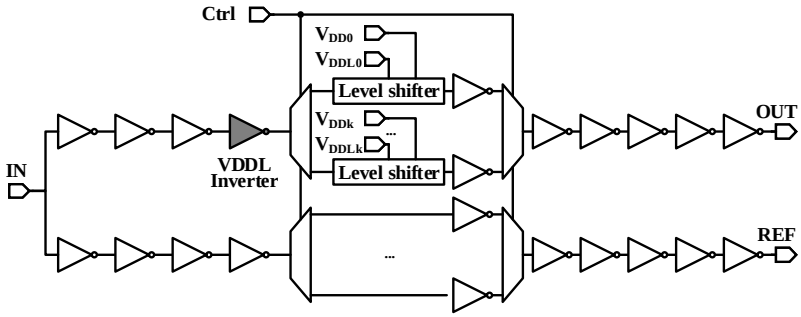
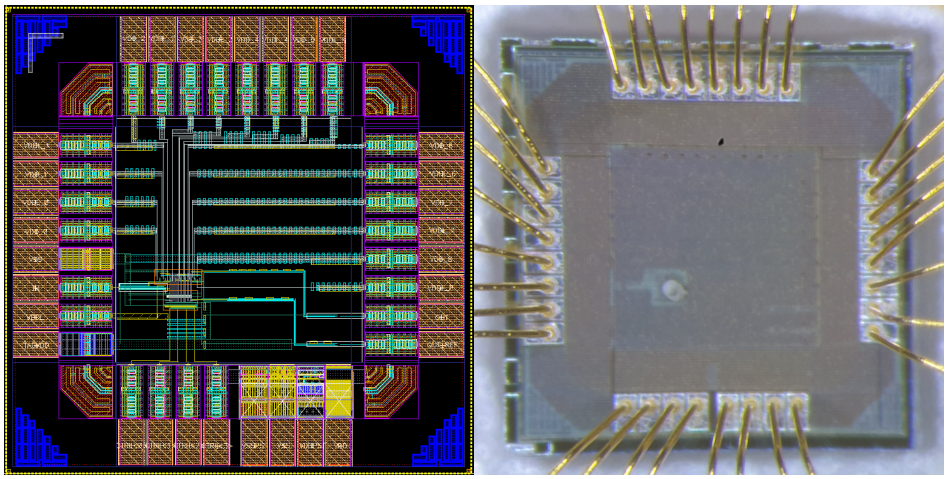


Fig. 4.9: Delay measurement circuit for testing

Table 4.5 displays measurement results and comparisons against recent level shifter circuits reported in the literature (Fig. 4.11). Note that the performance of the baseline circuits (B1-B5) are simulation results obtained from [40]. In measurements, all of the generated circuits (C1-C5) successfully perform level conversions. Measurement results show that our designs consume much smaller power consumption during conversion with similar or lower conversion delay. More specifically, our designs exhibit 2.6-4.7× lower total power consumption than the design with the lowest power consumption (B3) and 1.0-1.7× larger conversion delay than the fastest design (B5). In addition, our designs occupy 1.5-2.1× smaller area than the smallest design (B1). The power-delay product (PDP) is a metric commonly used for comparing level shifter circuits [24, 37, 40], and the generated circuits achieve 2.8-5.3× lower PDP than the



(a) (b)

Fig. 4.10: (a) test chip layout, (b) test chip micrography.

Table 4.5: Results of Measurement generated circuits

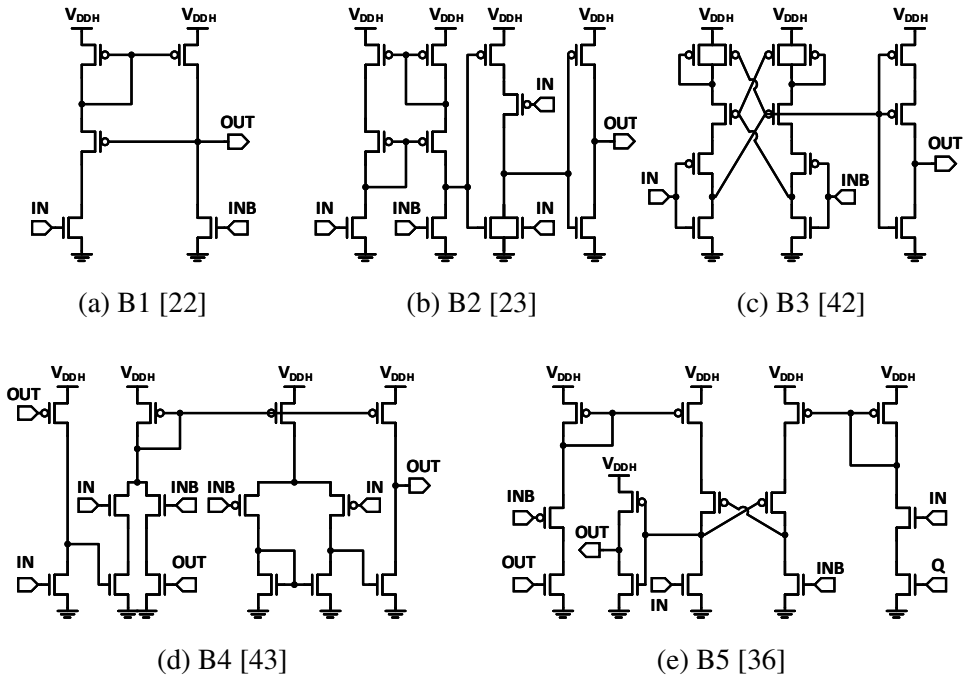
V_{DDL}=0.4V @ 1MHz						
	V_{DDLmin}(mV)	P_{total}(nW)	P_{static}(nW)	Delay(ns)	Area(μm^2)	PDP
B1*	400 / –	2654	0.98	61	69.1	161894
B2*	370 / –	584	0.18	36	74.4	21024
B3*	380 / –	290	0.22	54	99.8	15660
B4*	370 / –	320	0.13	35	103.5	11200
B5*	360 / –	327	0.13	31	120.9	10137
C1	320 / 42	69.8	0.44	51.6	40.3	3604
C2	280 / 41	85.4	0.47	30.7	39.2	2620
C3	310 / 34	113.2	0.45	31.2	46.3	3533
C4	320 / 70	76.3	1.01	39.2	33.1	2991
C5	270 / 48	62.0	0.95	30.8	32.4	1911

* Simulation results reported in [40]

baseline circuits.

V_{DDLmin} represents the minimum input voltage that a level shifter can convert to a high voltage signal. V_{DDLmin} was first measured for the input with 1MHz frequency. Generated circuits (C1-C5) achieve 320mV or lower V_{DDLmin}, outperforming baseline circuits. To determine the lowest possible voltage that the level shifters could handle, we also experimented with a 100Hz input signal and checked if the output shows full swing. In this case, the generated level shifters achieve significantly lower V_{DDLmin} less than 100mV.

Although C1-C5 performed better compared to the baseline, there are some points to consider. First of all, in the case of C1, the input range does not appear as a full range because the level shifter targeted in this experiment set 0.4V as the input voltage. When V_{DDL} rises above a certain voltage by input connected to M6, the voltage stops. Next, there is a case where an unnecessary transistor is inserted into the circuit. M5 of C3 does not play any role because it is directly connected to V_{DDH}. In this case, area can be wasted. Finally, as a feedback structure is formed by the output voltage in the circuit, a part vulnerable to noise occurs. For example, in the case of C5, if under noise occurs in the net between M7 and M6 while the input enters 0 and the output is maintained at 0, there is a possibility that the static current may increase. However, in the case of problems except the insertion of unnecessary transistors, it seems that it can be sufficiently supplemented by configuring the metric to be considered during the generation and optimization process.



(a) B1 [22] (b) B2 [23] (c) B3 [42]

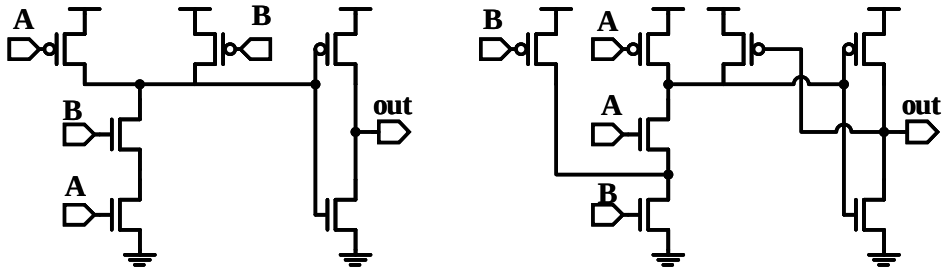
(d) B4 [43] (e) B5 [36]

Fig. 4.11: Baseline level shifter designs from prior work.

4.5 Applicability of Topology Generator

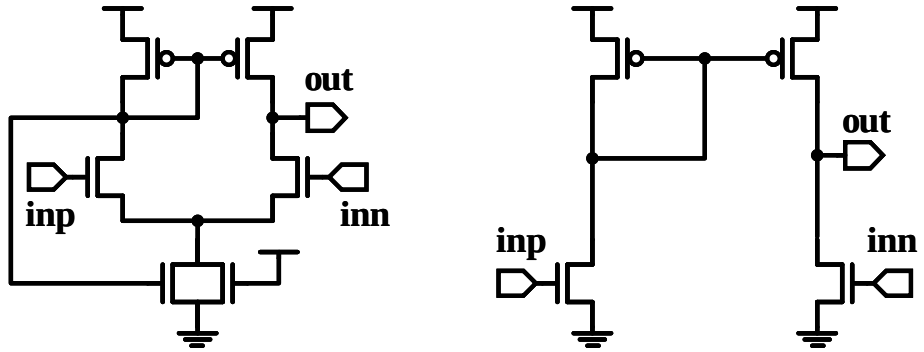
We conduct further experiments to observe if the proposed topology generator could be used for designing other types of circuits. For experiments, the topology generator is tested on both digital (AND gate) and analog (differential amplifier) circuits. In both cases, the algorithm starts with a P-channel MOSFET & N-channel MOSFET pair as the initial offspring and an initial population size of 600. For AND gate, the population evolves for 300 generations. We use a minimum-sized transistor with 180nm channel length and 220nm channel width as a weak device. Medium and strong devices have 2× and 4× larger channel width, respectively. The topology generator successfully produces a standard AND gate composed of a NAND gate and an inverter as shown in Fig. 4.12(a). The left part of the circuit in Fig. 4.12(b) is similar to a standard NAND gate, but the output is not fully pulled up since one of the PMOS devices is

connected to an internal node. However, the additional PMOS keeper fully pulls up the output node, providing a rail-to-rail output.



(a) (b)

Fig. 4.12: AND gates generated by topology generator.



(a) (b)

Fig. 4.13: Differential amplifiers generated by topology generator.

For amplifier design, the population evolves for 600 generations. Since analog circuits often require proper biasing, a bias node that supplies a DC voltage is introduced in the algorithm. In addition, five sizing options are used for topology generation. We use a transistor with 720nm channel length and 220nm channel width as a baseline. Two stronger devices have 2× and 4× larger channel width, respectively, whereas two weaker devices have 2× and 4× larger channel length, respectively. The topology generator successfully generates circuit topologies that are similar to widely used

amplifier circuits. The amplifier circuit in Fig. 4.13(a) is a self-biased 5T OTA (Operational Transconductance Amplifier) circuit [44], and the circuit in Fig. 4.13(b) is a low-voltage pseudo-differential amplifier [45, 46].

Chapter 5. Conclusion

In this work, we proposed an automatic circuit design framework for level shifter circuits. To design a circuit without pre-constructed building blocks and prior knowledge, the framework implements a two-step design process using the topology generator and the circuit optimizer. We first propose a new graph-based circuit representation, and the topology generator employs an evolutionary algorithm to search for possible circuit topologies quickly, considering the given design constraints. Then, the circuit optimizer utilizes reinforcement learning to fine-tune the size of each transistor, where we adopt various algorithmic optimizations such as multi-agent training, process variation aware optimization, multi-update, and episode early stopping to improve sample efficiency. In experiments, the framework was applied to designing level shifter circuits. The topology generator produced novel level shifter topologies, and they are successfully optimized by the circuit optimizer. Fabricated in a 180nm CMOS process, the test chip demonstrates that the automatically designed circuits achieve 2.8-5.3× lower PDP than manually designed level shifter circuits reported in the literature.

Bibliography

[1] L. Lavagno, L. Scheffer, and G. Martin, *EDA for IC implementation, circuit design, and process technology*. CRC press, 2016.

[2] O. Aaserud and I. R. Nielsen, "Trends in current analog design-a panel debate," *Analog Integrated Circuits and Signal Processing*, vol. 7, no. 1, pp. 5–9, 1995.

[3] M. C. Golumbic, A. Mintz, and U. Rotics, "An improvement on the complexity of factoring read-once Boolean functions," *Discret. Appl. Math.*, vol. 156, pp. 1633–1636, may 2008.

[4] V. N. Possani, V. Callegaro, A. I. Reis, R. P. Ribas, F. De Souza Marques, and L. S. Da Rosa, "Graph-Based Transistor Network Generation Method for Supergate Design," *IEEE Trans. VLSI Syst.*, vol. 24, pp. 692–705, feb 2016.

[5] H. Y. Koh, C. H. Séquin, and P. R. Gray, "OPASYN: A Computer for CMOS Operational Amplifiers," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 9, no. 2, pp. 113–125, 1990.

[6] W. Kruiskamp and D. Leenaerts, "DARWIN: CMOS opamp synthesis by means of a genetic algorithm," in *Proc. 32nd Annu. ACM/IEEE Des. Autom. Conf.*, pp. 139–144, 2002.

[7] T. McConaghy, P. Palmers, M. Steyaert, and G. G. Gielen, "Variation-aware structural synthesis of analog circuits via hierarchical building blocks and structural homotopy," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, pp. 1281–1294, jan 2009.

[8] M. Meissner and L. Hedrich, "FEATS: Framework for explorative analog topology synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, pp. 213–226, feb 2015.

[9] Z. Zhao and L. Zhang, “Graph-grammar-based analog circuit topology synthesis,” in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 1–5, 2019.

[10] J. R. Koza, F. Dunlap, F. H. Bennett, M. A. Keane, J. Lohn, and D. Andre, “Automated synthesis of computational circuits using genetic programming,” in *Proc. IEEE Int. Conf. Evol. Comput.*, pp. 447–452, 1997.

[11] J. D. Lohn and S. P. Colombano, “A Circuit Representation Technique for Automated Circuit Design,” *IEEE Trans. Evol. Comput.*, vol. 3, no. 3, pp. 205–219, 1999.

[12] Y. Sapargaliyev and T. G. Kalganova, “Unconstrained Evolution of Analogue Computational “QR” Circuit with Oscillating Length Representation,” in *Proc. Int. Conf. Evolvable Syst.*, pp. 1–10, sep 2008.

[13] J. Slez´ak, S. Slez´ak, and J. P. Zela, “Evolutionary Synthesis of Cube Root Computational Circuit Using Graph Hybrid Estimation of Distribution Algorithm,” *Radioengineering*, vol. 23, no. 1, p. 549, 2014.

[14] B. Liu, Q. Zhang, F. V. Fernandez, and G. G. Gielen, “An efficient evolutionary algorithm for chance-constrained bi-objective stochastic optimization,” *IEEE Trans. Evol. Comput.*, vol. 17, pp. 786–796, dec 2013.

[15] P. P. Prajapati and M. V. Shah, “Two stage CMOS operational amplifier design using particle swarm optimization algorithm,” in *Proc. IEEE UP Sect. Conf. Electr. Comput. Electron.*, pp. 1–5, 2015.

[16] R. A. Thakker, M. S. Baghini, and M. B. Patil, “Low-Power Low-Voltage analog circuit design using hierarchical particle swarm optimization,” in *Proc. Int. Conf. VLSI Des.*, pp. 427–432, 2009.

[17] W. Lyu, P. Xue, F. Yang, C. Yan, Z. Hong, X. Zeng, and D. Zhou, "An efficient Bayesian optimization approach for automated optimization of analog circuits," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 65, pp. 1954–1967, jun 2018.

[18] B. He, S. Zhang, F. Yang, C. Yan, D. Zhou, and X. Zeng, "An Efficient Bayesian Optimization Approach for Analog Circuit Synthesis via Sparse Gaussian Process Modeling," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2020.

[19] H. Wang, J. Yang, H.-S. Lee, and S. Han, "Learning to Design Circuits," *arXiv*, 2018.

[20] K. Settaluri, A. Haj-Ali, Q. Huang, K. Hakhamaneshi, and B. Nikolic, "AutoCkt: Deep Reinforcement Learning of Analog Circuit Designs," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, pp. 490–495, 2020.

[21] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "GCN-RL Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks and Reinforcement Learning," in *Des. Autom. Conf.*, pp. 1–6, 2020.

[22] S. Lutkemeier and U. Ruckert, "A subthreshold to above-threshold level shifter comprising a Wilson current mirror," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 57, no. 9, pp. 721–724, 2010.

[23] S. C. Luo, C. J. Huang, and Y. H. Chu, "A wide-range level shifter using a modified wilson current mirror hybrid buffer," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 61, no. 6, pp. 1656–1665, 2014.

[24] S. Kabirpour and M. Jalali, "A power-delay and area efficient voltage level shifter based on a reflected-output wilson current mirror level shifter," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 67, no. 2, pp. 250–254, 2020.

[25] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, 2002.

[26] J. Yosinski and J. Clune, "Evolving robot gaits in hardware: the HyperNEAT generative encoding vs. parameter optimization," *Eur. Conf. Artif. Life*, pp. 890–897, 2011.

[27] P. Verbancsics and K. O. Stanley, "Constraining connectivity to encourage modularity in HyperNEAT," *Genet. Evol. Comput. Conf.*, p. 1483, 2011.

[28] Tomáš Kocmánek, *HyperNEAT and Novelty Search*. PhD thesis, Czech Technical University in Prague, 2015.

[29] B. Jolley and A. Channon, "Toward evolving robust, deliberate motion planning with HyperNEAT," *IEEE Symp. Ser. Comput. Intell.*, vol. 2018-Janua, pp. 1–8, 2018.

[30] J. Merrild, M. A. Rasmussen, and S. Risi, "HyperNTM: Evolving Scalable Neural Turing Machines Through HyperNEAT," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10784 LNCS, pp. 750–766, 2018.

[31] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv*, sep 2015.

[32] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*, pp. 387–395, PMLR, 2014.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[34] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributed Distributional Deterministic Policy Gradients," *arXiv*, apr 2018.

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv*, jul 2017.

[36] S. R. Hosseini, M. Saberi, and R. Lotfi, "A low-power subthreshold to above-threshold voltage level shifter," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 61, pp. 753–757, oct 2014.

[37] S. Kabirpour and M. Jalali, "A Low-Power and High-Speed Voltage Level Shifter Based on a Regulated Cross-Coupled Pull-Up Network," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 66, no. 6, pp. 909–913, 2019.

[38] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, pp. 1310–1318, 2013.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 1026–1034, 2015.

[40] S. R. Hosseini, M. Saberi, and R. Lotfi, "A High-Speed and Power-Efficient Voltage Level Shifter for Dual-Supply Applications," *IEEE Trans. VLSI Syst.*, vol. 25, no. 3, pp. 1154–1158, 2017.

[41] R. Lotfi, M. Saberi, S. R. Hosseini, A. R. Ahmadi-Mehr, and R. B. Staszewski, "Energy-Efficient Wide-Range Voltage Level Shifters Reaching 4.2 fJ/Transition," *IEEE Solid-State Circuits Lett.*, vol. 1, no. 2, pp. 34–37, 2018.

[42] M. Lanuzza, P. Corsonello, and S. Perri, "Fast and wide range voltage conversion in multisupply voltage designs," *IEEE Trans. VLSI Syst.*, vol. 23, pp. 388–391, feb 2015.

[43] Y. Osaki, T. Hirose, N. Kuroki, and M. Numa, "A low-power level shifter with logic error correction for extremely low-voltage digital CMOS LSIs," *IEEE J. Solid-State Circuits*, vol. 47, no. 7, pp. 1776–1783, 2012.

[44] B. A. Chappell, T. I. Chappell, S. E. Schuster, H. M. Segmuller, J. W. Allan, R. L. Franch, and P. J. Restle, "Fast cmos ecl receivers with 100-mv worst-case sensitivity," *IEEE J. Solid-State Circuits*, vol. 23, no. 1, pp. 59–67, 1988.

[45] C. J. A. Gomez, H. Klimach, E. Fabris, and O. E. Mattia, "High psrr nano-watt mos-only threshold voltage monitor circuit," in *IEEE Symp. Integr. Circuits Syst. Design (SBCCI)*, pp. 1–6, IEEE, 2015.

[46] A. Shankar, J. Silva-Martínez, and E. Sánchez-Sinencio, "A low voltage operational transconductance amplifier using common mode feedforward for high frequency switched capacitor circuits," in *IEEE Int. Symp. Circuits Syst.*, vol. 1, pp. 643–646, IEEE, 2001.

초 록

유전알고리즘 및 강화학습을 사용한 고속 회로 설계 자동화 프레임워크

홍 지 우
지능정보융합학과
서울대학교 대학원

설계 자동화는 대규모 디지털 시스템을 가능하게 하는 핵심 요소이지만 트랜지스터 수준에서 회로 설계 프로세스를 자동화하는 것은 여전히 어려운 과제로 남아 있습니다. 최근 연구에서는 아날로그 앰프와 같은 비교적 작은 회로에서 최적의 성능을 보이는 트랜지스터 크기를 찾기 위해 deep learning 알고리즘을 활용할 수 있다고 말합니다. 그러나 이러한 접근 방식은 주어진 설계 constraint를 충족하는 다른 회로 구조 탐색에 적용하기 어렵습니다. 본 연구에서는 성능과 신뢰성을 고려하여 각 트랜지스터의 크기를 최적화할 뿐만 아니라 처음부터 실용적인 회로 구조를 생성할 수 있는 자동 회로 설계 framework를 제안합니다. 우리는 framework를 사용하여 level shifter 회로를 설계했으며 실험 결과는 프레임워크가 새로운 level shifter 회로 토폴로지를 생성하고 자동으로 최적화된 설계가 인간 전문가가 설계한 선행 기술보다 2.8-5.3배 더 낮은 PDP를 달성한다는 것을 보여줍니다.

주요어: 회로 설계 자동화, 딥러닝, 유전 알고리즘, 레벨 시프터, 강화학습.
학번: 2019-21653