MiniTNtk: An Exact Synthesis-based Method for Minimizing Transistor Network

Weihua Xiao[1], Shanshan Han[1], Yue Yang[1], Shaoze Yang[1], Cheng Zheng[1], Jingsong Chen[2], Tingyuan Liang[2], Lei Li[2], Weikang Qian[1,3]
[1]UM-SJTU Joint Inst. and [3]MoE Key Lab of AI, Shanghai Jiao Tong University, China; [2]Huawei Technologies Co., Ltd., China
Emails: {019370910014, sshan1996, yue.yang, priest-yang, Alex_zc}@sjtu.edu.cn,
{chenjingsong5, liangtingyuan1, lilei291}@huawei.com, qianwk@sjtu.edu.cn

***Abstract*—Transistor network minimization is an important step in designing new standard cells. Existing methods for minimizing transistor networks all rely on some heuristic techniques. Hence, there is still room for further improvement. In this work, we propose MiniTNtk, an exact synthesis-based method for minimizing transistor networks. It models the generation of the transistor network for a Boolean function as a Boolean satisfiability (SAT) problem and can return a transistor network with the fewest transistors. Furthermore, sometimes, it is necessary to limit the number of transistors in series. We propose an extension of MiniTNtk for minimizing the transistor network under a bound on the number of transistors in series. The experimental results showed that MiniTNtk is the first method that achieves the optimal transistor networks for a set of Boolean functions with known optimal solutions to the best of our knowledge. Additionally, compared with related works, MiniTNtk reduces the number of transistors by up to 9.39% over all 4-input P-class representative functions. Moreover, the experiment on a complex Boolean function demonstrated the high efficiency of MiniTNtk.**

***Index Terms*—transistor network, Boolean function, exact synthesis, SAT problem**

I. Introduction

As transistors approach their physical limit, the further scaling down witnesses diminishing returns in power consumption reduction [1]. To overcome this challenge, advanced standard cell design techniques are being actively explored [2, 3, 4]. Among them, one promising solution is to introduce new complex gates into a standard cell library, which are obtained by merging multiple basic gates from the library [3]. An essential step in designing a complex gate is generating a transistor network implementing the Boolean function of the gate. Thus, it is desired to have a computer-aided design tool for generating a transistor network with minimized cost for a given Boolean function. We target at this problem in this work. A typical measure on the cost of a transistor network is its number of transistors. We also adopt this measure, and consequently, we aim at synthesizing a transistor network with the fewest transistors, which we refer to as *minimizing a transistor network* hereafter. Note that the technique of minimizing transistor networks can be applied not only to CMOS technology, but also to new technologies, such as FinFET and carbon nanotube technologies [5].

Many different methods have been proposed for minimizing transistor networks. They can be divided into two categories: Boolean function factorization-based [6, 7, 8, 9] and graph-based [10, 11, 12, 13, 5]. Factorization-based methods convert a given target Boolean function into a factored form with a minimized number of literals by Boolean factorization. Then, the factored form is directly converted into a series-parallel (SP)-only transistor network, such that each literal corresponds to a transistor in the transistor network. However, the Boolean factorization-based method can only generate SP transistor networks [8], while properly taking advantage of non-series-parallel (NSP) structure can save more transistors. Moreover, the size of the synthesized transistor network highly depends on the initial form of the Boolean function.

On the other hand, graph-based methods transform the transistor network minimization into the simplification over a graph based on some heuristic techniques. Some works apply reduced ordered binary decision diagrams (ROBDDs) to minimize transistor networks [11, 14]. These methods are independent of the input form of a Boolean function. However, the performance of ROBDD-based simplification heavily depends on the variable ordering, which is hard to determine [15]. Another graph-based method starts from a sum-of-products (SOP) expression of the target Boolean function and constructs its transistor network one product term by another [12]. In order to minimize the transistor network, it proposes an NSP structure-based technique, which can find a construction using the fewest transistors for each product term. However, both the initial SOP form and the processing order of the product terms in the SOP can affect the final result. Some other graph-based methods [13, 5] first convert the given Boolean function into a graph and then reduce the number of its edges through a greedy *edge sharing* technique. However, the order of performing edge sharing can influence the reduction of transistors. Moreover, in some cases, limiting the number of transistors in series is critical for the performance of the transistor network, but such a requirement is less considered [5].

In this paper, leveraging the power of modern satisfiability (SAT) solvers, we propose MiniTNtk, a novel exact synthesis-based method for minimizing transistor network. It models the generation of the transistor network for a Boolean function as a Boolean SAT-based exact synthesis problem. The basic version of it is independent of the Boolean function representation and can produce a transistor network with the fewest transistors. Additionally, as it is necessary to limit the number of transistors in series when considering the performance of the transistor network, we propose an extension of MiniTNtk for minimizing the transistor network under a bound on the number of transistors in series. Furthermore, we propose two techniques to accelerate the solving of the SAT-based exact synthesis problem. The experimental results showed that MiniTNtk is the first one that achieves the optimal transistor networks for a set of Boolean functions with known optimal solutions to the best of our knowledge. Additionally, compared to related works, MiniTNtk can reduce the number of transistors by up to 9.39% over all 4-input P-class representative functions. Moreover, the experiment on a complex Boolean function demonstrated the high efficiency of MiniTNtk by using our proposed acceleration techniques.

The rest of the paper is organized as follows. Section II introduces the preliminaries. Section III presents our proposed exact synthesis method for transistor networks. Section IV shows the flow of MiniTNtk. Section V shows the experimental results. Finally, Section VI

This work is supported in part by the National Key R&D Program of China under Grants 2021ZD0114701 and 2020YFB2205501 and the National Natural Science Foundation of China under Grants T2293700 and T2293701. Corresponding author: Weikang Qian.

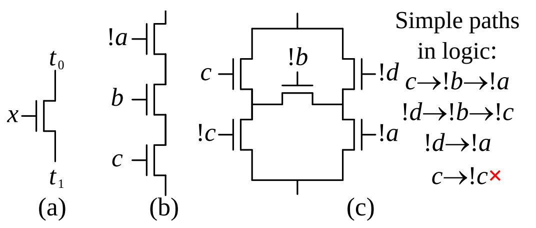

Fig. 1. Examples of transistor networks: (a) a single transistor; (b) a series of transistors; (c) a NSP transistor network.

concludes the paper.

II. Preliminary

A. Boolean Functions and Transistor Networks

An n-input Boolean function f is defined as $f(X) : B^n \to B$, where $B = \{0,1\}$ and X is the variable set $\{x_0, x_1, \cdots, x_{n-1}\}$, in which x_i $(0 \le i \le n-1)$ is the i-th input. A *literal* is a variable or its complement, *e.g.*, x_i or $!x_i$, and there are $2n$ literals of f in total, denoted as the literal set $L_f = \{x_0, x_1, \ldots, x_{n-1}, !x_0, !x_1, \ldots, !x_{n-1}\}$. A Boolean function can be represented in different forms, such as SOP and product-of-sums, which are made of three operations, *AND* ($\wedge$), *OR* ($\vee$), and *NOT* (!), and literals from L_f. In this paper, we only use the SOP form. If there only exists literal x_i (*resp.* $!x_i$) for variable x_i in an SOP, variable x_i is a *positive* (*resp. negative*) *unate* variable of the SOP. Next, we introduce some basic definitions of a Boolean function:

- *Index of an input pattern*: the index k $(0 \le k \le 2^n - 1)$ of an input pattern of an n-input Boolean function is the binary radix value encoded by the pattern. The k-th input pattern is denoted as p_k;
- *Onset*: onset is the set of indices of all input patterns that make the Boolean function yield 1. It is denoted as S_{on}. An input pattern p_k with $k \in S_{on}$ is called an *onset input pattern*;
- *Offset*: offset is the set of indices of all input patterns that make the Boolean function yield 0. It is denoted as S_{off}. An input pattern p_k with $k \in S_{off}$ is called an *offset input pattern*;
- *Literal pattern*: the literal pattern lp_k of an input pattern p_k is a vector composed of the values of all literals in the set L_f under the input pattern p_k. For example, the literal pattern of an input pattern $(a,b) = (0,1)$ is $(a, b, !a, !b) = (0,1,1,0)$.

In modern CMOS design, which is the focus of this work, each Boolean function is implemented by two logically complementary transistor networks: a pull-up PMOS transistor network and a pull-down NMOS transistor network. Note that our proposed method can be used for both pull-up and pull-down networks, and without loss of generality, we illustrate our method on pull-down networks in what follows. Each transistor in a transistor network is modeled as a *switch* with three terminals: one *control terminal* controlled by an input signal and two *contact terminals* connected with other transistors. If the input signal is 1, the switch is closed, and thus, current can flow between the two contact terminals. Otherwise, it is open. The terms transistor and switch are used interchangeably in the remaining paper. The input signal of a switch should be one of the $2n$ literals of the given n-input Boolean function. Thus, in what follows, the input signal of a switch is also called *the literal of a switch*. Fig. 1(a) shows an example of a transistor with the control terminal controlled by signal x and two contact terminals t_1 and t_2. In a transistor network, there are two special contact terminals: one, called the *source terminal*, is connected to *GND* or *VDD*, while the other, called the *sink terminal*, is connected to the output of the network.

A basic topology of a transistor network is a series connection of a set of transistors. Its corresponding Boolean function is the *AND* of all literals of the transistors, *i.e.*, a product term. For example, the network in Fig. 1(b) corresponds to the Boolean function $!a \wedge b \wedge c$, where the top contact terminal of the switch with literal $!a$ and the bottom contact terminal of the switch with literal c are the source and sink terminals of the transistor network, respectively.

There are two approaches for deriving the Boolean function of a transistor network: the graph-based and the behavior-based [16]. In the graph-based approach, it first gets all *simple paths* (*i.e.*, paths without cycles) between the source and sink terminals and then performs a logical *OR* operation on the Boolean function of each simple path (*i.e.*, a product term) to obtain the final Boolean function. For example, there are four simple paths in the transistor network shown in Fig. 1(c). Since the path $c \to !c$ does not exist logically, the final Boolean function is $(c \wedge !b \wedge !a) \vee (!d \wedge !b \wedge !c) \vee (!d \wedge !a)$.

The behavior-based method checks each input pattern in turn to derive the Boolean function. An input pattern is an onset one if a simple path exists between the source and sink terminals of the transistor network under the configuration of it; otherwise, it is an offset one. For example, the input pattern $(a,b,c,d) = (0,0,1,0)$ is an onset input pattern in Fig. 1(c) as a simple path $c \to !b \to !a$ can be found in the network under the configuration of it.

B. SAT-based Exact Synthesis

Exact synthesis is a technique used in logic synthesis for generating a logic representation meeting some specifications, such as generating a logic network for implementing a given Boolean function f with exactly r gates [17]. The exact synthesis can be adopted to synthesize the optimal logic network for a target Boolean function by 1) initializing r to zero and solving the exact synthesis problem and 2) increasing r one by one until finding the first value r for which the exact synthesis is solved successfully. To solve an exact synthesis problem, it is transformed into a Boolean SAT problem that is further solved by a SAT solver [18]. The SAT-based exact synthesis has been applied into different fields, such as logic synthesis [17] and approximate computing [19]. The key to apply the technique is to transform an exact synthesis problem in a specific field into a SAT problem, which is formulated in conjunctive normal forms (CNFs). In this paper, we apply the SAT-based exact synthesis to solve the transistor network minimization problem for the first time. It should be noted that the exact synthesis for a transistor network differs significantly from that for a gate-level network. For example, each transistor should be bound with a unique literal, and a current can flow in either direction through a transistor.

III. Exact Synthesis for Transistor Network

MiniTNtk is based on solving the following *exact synthesis problem for transistor network*:

Definition 1. *Given a Boolean function $f : B^n \to B$ and exactly r transistors, determine whether a transistor network with exactly r transistors exists implementing the given Boolean function f. If so, also return such a transistor network.*

This section will show how to solve the above problem. Our solution transforms it into the problem of constructing a special graph called generalized switching graph, which will be described in Section III-A. Then, we model the construction process of a generalized switching graph as a SAT problem in Section III-B.

In order to ensure that the generated transistor network implements the given Boolean function, additional constraints are added into the above SAT problem, which will be described in Section III-C. Table I lists the major variables used in the following sections.

TABLE I
MAJOR VARIABLES USED IN THE SAT-BASED EXACT SYNTHESIS.

n: number of inputs of the given Boolean function;
r: number of transistors in the constructed transistor network;
L_f: the literal set of Boolean function f
p_k $(0 \leq k \leq 2^n - 1)$: the k-th input pattern;
lp_k $(0 \leq k \leq 2^n - 1)$: the literal pattern of p_k;
E_s: the set of all switching edges in a generalized switching graph;
E_{pi}: the set of all possible internal edges in a blank generalized switching graph;
$x_{i,j}$ $((i,j) \in E_{pi})$: if $x_{i,j} = 1$, the possible internal edge (i,j) exists; otherwise, no internal edge exists between nodes i and j;
$l_{i,j,v}$ $((i,j) \in E_s, 0 \leq v \leq 2n-1)$: if $l_{i,j,v} = 1$, the switching edge between nodes i and j is bound with literal $L_f[v]$;
$c_{k,i,j}$ $(0 \leq i,j \leq 2r+1, i \neq j)$: if the directed edge $i \to j$ exists in the directed switching graph under the configuration of onset input pattern p_k, $c_{k,i,j} = 1$; otherwise, $c_{k,i,j} = 0$;
$f_{k,i,j}$ $(0 \leq i,j \leq 2r+1, i \neq j)$: if the edge $i \to j$ is on the found simple path under the configuration of onset input pattern p_k, $f_{k,i,j} = 1$; otherwise, $f_{k,i,j} = 0$;
$t_{m,i,j}$ $(0 \leq i < j \leq 2r+1)$: if the edge (i,j) exists in the generalized switching graph under the configuration of offset input pattern p_m, $t_{m,i,j} = 1$; otherwise, $t_{m,i,j} = 0$;
$s_{m,i}$ $(0 \leq i \leq 2r+1)$: if node i belongs to the first (*resp.* second) set of nodes of the node separation under the configuration of offset input pattern p_m, $s_{m,i} = 0$ (*resp.* 1).

A. Generalized Switching Graph

In order to generate a satisfying transistor network for the exact synthesis problem shown in Definition 1, we need to determine 1) how the transistors are connected, and 2) which literal should be bound to each transistor, so that the transistor network realizes the given Boolean function. In this section, we propose a concept called *generalized switching graph*, which aids the answer to the above two questions. It is defined as follows:

Definition 2. *A **generalized switching graph** with r transistors is denoted as $SG(r) = G(V,E)$. The set of nodes V consists of $2r$ nodes representing the contact terminals of the r transistors and 2 nodes representing source and sink terminals, respectively. The set of edges E consists of two types of edges as follows:*

- ***switching edge**: represents a transistor with its corresponding literal. Similar to a transistor, whether it exists in the graph depends on a given input pattern: if the literal under the input pattern is 1, it exists; otherwise, it does not;*
- ***internal edge**: represents the connection between two contact terminals of two different transistors or the connection between a contact terminal of a transistor and a source/sink terminal.*

The set of all switching edges is denoted as E_s.

In what follows, we denote the $(2r+2)$ nodes in the transistor network as nodes $0, 1, \ldots, (2r+1)$, where nodes 0 and $(2r+1)$ correspond to the source and sink terminals, respectively. Fig. 2(a) illustrates a generalized switching graph $SG(5)$ with 5 transistors, where the nodes are drawn as circles, and the literals of switching edges are labeled in the squares. For example, the edge between nodes 0 and 7 is an internal edge, and the edge between nodes 1 and 2 is a switching edge with the corresponding literal as c.

Each generalized switching graph has a corresponding transistor network. Once a generalized switching graph realizing the given Boolean function is found, we can convert it into its corresponding transistor network. The conversion exploits a definition called *node cluster* as follows.

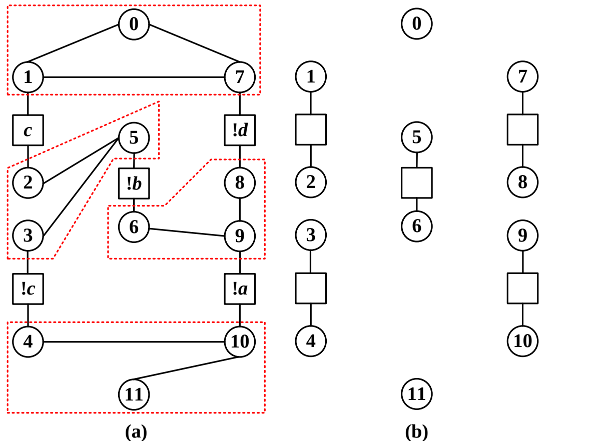

Fig. 2. Examples of (a) a generalized switching graph and (b) a blank generalized switching graph.

Definition 3. *A set of nodes of a generalized switching graph is a **node cluster** if each node in the set can reach any other node in the set through a simple path consisting of only internal edges.*

For all nodes in a node cluster, their corresponding contact/-source/sink terminals in the transistor network are shorted, and hence, these terminals can be merged into a single one in the transistor network. Thus, the graph conversion is to find all node clusters in a generalized switching graph and merge the nodes of each node cluster into a single terminal in the corresponding transistor network, which can be implemented by breadth first search. We omit the details due to space limit. In Fig. 2(a), each node cluster is shown in a red dotted box. For example, the node set $\{2,3,5\}$ is a node cluster. The final converted transistor network is shown in Fig. 1(c).

B. Modeling the Construction of Generalized Switching Graph

Given the relationship between the generalized switching graph and the transistor network, we transform the exact synthesis problem into constructing a generalized switching graph with r transistors realizing the given Boolean function. We start the construction from a *blank generalized switching graph* with r transistors, which is a special generalized switching graph with no internal edges and no corresponding literal of each switching edge. In other words, it only has switching edges that are bound with no literals. Fig. 2(b) illustrates a blank generalized switching graph with 5 transistors. There is no internal edge in it, and the square of each switching edge is blank, indicating that the literal of the switching edge is undetermined. We define a set called *possible internal egde set* E_{pi} for the blank generalized switching graph, which consists of all possible internal edges in the blank graph. From the perspective of the generalized switching graph, the exact synthesis problem for transistor network is transformed into determining 1) which possible internal edges in E_{pi} should actually exist and 2) the literal of each switching edge.

In the following, we denote an undirected edge between nodes i and j $(i < j)$ as (i,j) and a directed edge from node i to node j as $i \to j$. To formulate the construction of a generalized switching graph $SG(r)$ for an n-input Boolean function f as a SAT problem, we define two types of binary variables and a set of related constraints:

- *Internal edge variable* $x_{i,j}$ ($(i,j) \in E_{pi}$): if $x_{i,j} = 1$, the possible internal edge (i,j) actually exists; otherwise, nodes i and j are not connected by an internal edge;
- *Literal selection variable* $l_{i,j,v}$ ($(i,j) \in E_s$ and $0 \le v \le 2n-1$): if $l_{i,j,v} = 1$, the switching edge between node i and node j is bound with the literal $L_f[v]$;
- *Literal selection constraint*: we have to ensure that each switching edge is bound with a literal, which is formulated as a constraint using the literal selection variables as follows:

$$\sum_{v=0}^{2n-1} l_{i,j,v} = 1, \ \forall (i,j) \in E_s. \tag{1}$$

  Note that the above constraint is a cardinality constraint, which can be converted into two sets of CNFs [17].

Thus, the construction of a generalized switching graph is equivalent to determining the values of the above two types of variables. For example, the generalized switching graph in Fig. 2(a) is derived from the blank one in Fig. 2(b) by setting the corresponding value for each internal edge variable and each literal selection variable, *e.g.*, $x_{4,10} = 1$ and $l_{1,2,2} = 1$ assuming that the literal set L_f of f is $\{a, b, c, d, !a, !b, !c, !d\}$.

C. Modeling the Constraint Due to the Given Boolean Function

A valid generalized switching graph should implement the given Boolean function. In this section, we add additional constraints to satisfy this requirement. According to the behavior-based method for deriving the Boolean function described in Section II, we have to ensure that each onset (*resp.* offset) input pattern of the given Boolean function leads to the (*resp.* no) existence of a simple path connecting nodes 0 and $(2r+1)$ in the generated generalized switching graph (*i.e.*, source and sink terminals in the transistor network).

The following two subsections show how to model the constraints due to an onset and an offset input pattern, respectively.

1) Modeling the Constraint Due to an Onset Input Pattern: This section models the constraint due to an onset input pattern p_k ($k \in S_{on}$), which is to make sure that there exists a simple path between nodes 0 and $(2r+1)$ in the generalized switching graph under the switch configuration of the input pattern p_k. In what follows, if such a path is found, we call it *the found simple path* for simplicity.

As the generalized switching graph is an undirected graph, we first convert it into a *directed switching graph* for ease of finding a simple path by 1) replacing an undirected internal edge (i,j) with two directed edges $i \to j$ and $j \to i$, and 2) replacing an undirected switching edge (i,j) with literal x as two directed switching edges $i \to j$ and $j \to i$ both with literal x. Note that the two directed internal edges $i \to j$ and $j \to i$ have the same internal edge variable $x_{i,j}$, and the two directed switching edges $i \to j$ and $j \to i$ also have the same literal selection variable $l_{i,j,v}$.

Example 1. *Fig. 3(a) shows the directed switching graph derived from the generalized switching graph in Fig. 2(a) under an onset input pattern* $p_6 = (a,b,c,d) = (0,1,1,0)$. *It is obtained by removing the switching edges corresponding to the switches turned off under input pattern* p_6 *and then replacing each remaining undirected edge into two directed ones.*

Over the directed switching graph, we define two types of binary variables:

- *Onset edge existence variable* $c_{k,i,j}$ ($0 \le i,j \le 2r+1, i \ne j$): If the directed edge $i \to j$ exists in the directed switching graph under the configuration of input pattern p_k, $c_{k,i,j} = 1$; otherwise, $c_{k,i,j} = 0$. Note that there exist two onset edge existence variables for each pair of nodes in V;
- *On-path variable* $f_{k,i,j}$ ($0 \le i,j \le 2r+1, i \ne j$): if the edge $i \to j$ is on the found simple path under the configuration of input pattern p_k, $f_{k,i,j} = 1$; otherwise, $f_{k,i,j} = 0$. Similarly, each pair of nodes has two on-path variables.

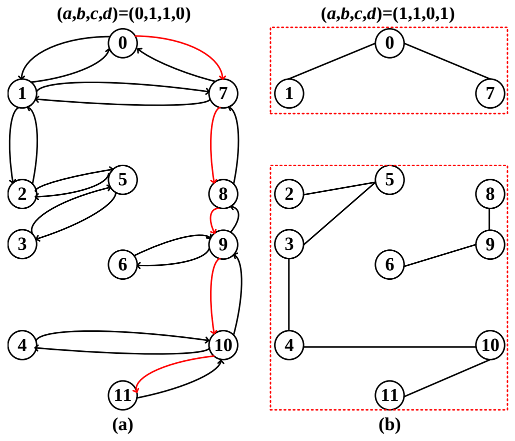

Fig. 3. Examples of modeling (a) an onset and (b) an offset input pattern.

Based on the two types of variables, we propose to use six constraints for modeling the existence of a simple path between nodes 0 and $(2r+1)$ under the onset input pattern p_k.

- *Onset existence constraint for a possible internal edge*: for a possible internal edge (i,j), the onset edge existence variables of its two directed edges $i \to j$ and $j \to i$ are both 1 if and only if edge (i,j) actually exists, *i.e.*, its internal edge variable equals 1. This constraint is formulated as:

$$(c_{k,i,j} \odot x_{i,j}) \land (c_{k,j,i} \odot x_{i,j}) = 1, \ \forall (i,j) \in E_{pi}, \tag{2}$$

  where $\odot$ is the logical *XNOR* operation.
- *Onset existence constraint for a switching edge*: for a switching edge, the onset edge existence variables of its two directed edges are 1 if and only if the value of the literal of the switching edge equals 1 under the input pattern p_k, *i.e.*, the switch is turned on under this input pattern. This constraint is formulated as:

$$\begin{aligned}&(c_{k,i,j} \odot \bigvee_{v=0}^{2n-1}(l_{i,j,v} \land lp_k[v]))\\ &\land(c_{k,j,i} \odot \bigvee_{v=0}^{2n-1}(l_{i,j,v} \land lp_k[v])) = 1, \ \forall (i,j) \in E_s.\end{aligned} \tag{3}$$

  In Eq. (3), $(l_{i,j,v} \land lp_k[v])$ represents whether the switching edge (i,j) is bound with the literal $L_f[v]$ and whether $L_f[v]$'s value $lp_k[v]$ equals 1 under the input pattern p_k. In other words, it represents whether the switching edge (i,j) exists if it is bound with the literal $L_f[v]$ under the input pattern p_k.
- *On-path necessary constraint*: a directed edge can be on the found simple path only when it exists, *i.e.*, the onset edge existence variable is 1. The constraint is formulated as:

$$!f_{k,i,j} \lor \ c_{k,i,j} = 1, \forall 0 \le i,j \le 2r+1, i \ne j. \tag{4}$$

- *Source terminal constraint*: exactly one directed edge starting from node 0 is on the found simple path. This is formulated as:

$$\sum_{j=1}^{2r+1} f_{k,0,j} = 1. \tag{5}$$

Moreover, no directed edge ending at node 0 is on the found simple path, formulated as:

$$f_{k,i,0} = 0,\ \forall 1 \le i \le 2r+1. \tag{6}$$

- *Sink terminal constraint*: exactly one directed edge ending at node $(2r+1)$ is on the found simple path, formulated as:

$$\sum_{i=0}^{2r} f_{k,i,2r+1} = 1. \tag{7}$$

Moreover, no edge starting from node $(2r+1)$ is on the found simple path, formulated as:

$$f_{k,2r+1,j} = 0,\ \forall 0 \le j \le 2r. \tag{8}$$

- *Simple path constraint*: This constraint is to ensure that the found simple path is valid by the definition of a simple path. Two requirements need to be satisfied: 1) if a node i ($i \ne 0, 2r+1$) is on the found simple path, there is exactly one edge starting from node i and one edge ending at node i on the found simple path; 2) if a node i ($i \ne 0, 2r+1$) is not on the simple path, there is no edge starting from or ending at node i on the found simple path. An equivalent form of the above two requirements is the following two requirements:
1) For each node other than nodes 0 and $(2r+1)$, there is at most one edge starting from it and at most one edge ending at it located on the found path, which can be formulated as:

$$\sum_{j:j\ne i} f_{k,i,j} \le 1, \sum_{j:j\ne i} f_{k,j,i} \le 1,\ \forall 1 \le i \le 2r. \tag{9}$$

2) For each node other than nodes 0 and $(2r+1)$, the number of edges starting from it on the found simple path should equal the number of edges ending at it on the found simple path, which can be formulated as:

$$\sum_{j:j\ne i} f_{k,i,j} = \sum_{j:j\ne i} f_{k,j,i},\ \forall 1 \le i \le 2r.$$

Given Eq. (9), the above equation can be transformed into the following one using the logical *OR* operation:

$$\bigvee_{j:j\ne i} f_{k,i,j} \odot \bigvee_{j:j\ne i} f_{k,j,i} = 1,\ \forall 1 \le i \le 2r. \tag{10}$$

If an assignment of variables $x_{i,j}$, $l_{i,j,v}$, $c_{k,i,j}$, and $f_{k,i,j}$ can be found satisfying Eqs. (1)–(10), a simple path connecting nodes 0 and $(2r+1)$ exists in the generalized switching graph under the configuration of input pattern p_k.

Example 2. *In Fig. 3(a), under the onset input pattern $p_6 = (a,b,c,d) = (0,1,1,0)$, the values of the two onset edge existence variables of each possible internal edge and each switching edge that do not exist, are 0 according to Eqs. (2) and (3). For example, $c_{6,2,3} = c_{6,3,2} = 0$ and $c_{6,3,4} = c_{6,4,3} = 0$. The on-path variables of these edges should also equal 0 according to Eq. (4). Since input pattern p_6 is an onset input pattern, we can find a solution satisfying Eqs. (5)–10 with $f_{6,0,7}, f_{6,7,8}, f_{6,8,9}, f_{6,9,10}$, and $f_{6,10,11}$ as 1, and the other on-path variables as 0. The found simple path corresponding to this solution is highlighted in red in Fig. 3(a).*

2) Modeling the Constraint Due to an Offset Input Pattern: This section models the constraint due to an offset input pattern p_m ($m \in S_{\mathit{off}}$), which is to ensure that no path exists between nodes 0 and $(2r+1)$ in the generalized switching graph under the switch configuration of the input pattern p_m. We first introduce the following definition.

Definition 4. *A **node separation** is a partition of the node set V into two sets with node 0 in the first set and node $(2r+1)$ in the second set such that there exists no edge between a node in the first set and a node in the second.*

An example of a node separation is shown in Fig. 3(b), where two sets of separated nodes are shown in two red dotted boxes.

By Definition 4, we can easily see that there exists no path between nodes 0 and $(2r+1)$ if and only if we can find a node separation of the node set V. Thus, to model the constraint due to an offset input pattern, we only need to make sure that there exists a node separation in the generalized switching graph under the switch configuration of the offset input pattern. In what follows, if such a node separation is found, we call it *the found node separation* for simplicity.

Over the generalized switching graph, we first introduce two types of binary variables.

- *Offset edge existence variable* $t_{m,i,j}$ ($0 \le i < j \le 2r+1$): if the edge (i,j) exists in the generalized switching graph under the configuration of offset input pattern p_m, then $t_{m,i,j} = 1$; otherwise, $t_{m,i,j} = 0$.
- *Node separation variable* $s_{m,i}$ ($0 \le i \le 2r+1$): if node i belongs to the first (*resp.* second) set of nodes of the found node separation under the configuration of input pattern p_m, $s_{m,i} = 0$ (*resp.* 1).

Based on the two types of variables, we introduce four constraints for modeling the existence of a node separation under offset input pattern p_m.

- *Offset existence constraint for a possible internal edge*: for a possible internal edge (i,j), the offset edge existence variable is 1 if and only if its internal edge variable equals 1. This constraint is formulated as:

$$t_{m,i,j} \odot x_{i,j} = 1,\ \forall (i,j) \in E_{pi}. \tag{11}$$

- *Offset existence constraint for a switching edge*: for a switching edge, the offset edge existence variable is 1 if and only if the literal of the switching edge equals 1. This constraint is formulated as:

$$t_{m,i,j} \odot \bigvee_{v=0}^{2n-1} (l_{i,j,v} \wedge lp_m[v]) = 1,\ \forall (i,j) \in E_s. \tag{12}$$

- *Constraint on source/sink terminal*: by Definition 4, we have

$$s_{m,0} = 0, s_{m,2r+1} = 1. \tag{13}$$

- *Node separation constraint*: if two nodes i and j have different values of their node separation variables $s_{m,i}$ and $s_{m,j}$, then edge (i,j) should not exist under the configuration of input pattern p_m, *i.e.*, $t_{m,i,j} = 0$. The constraint is formulated as:

$$(s_{m,i} \odot s_{m,j}) \vee\ !t_{m,i,j} = 1,\ \forall 0 \le i,j \le 2r+1, i \ne j. \tag{14}$$

If an assignment of variables $x_{i,j}$, $l_{i,j,v}$, $t_{m,i,j}$, and $s_{m,i}$ can be found satisfying Eqs. (1) and (11)–(14), no path exists in the corresponding switching graph under the configuration of pattern p_m.

IV. Flow of MiniTNtk

In this section, we describe the flow of MiniTNtk, which is shown in Algortihm 1. It takes an n-input Boolean function f as input and outputs a transistor network *tntk* implementing f. It has two modes determined by another binary input variable, *SeLimit*. If *SeLimit* is 0 (*resp.* 1), it is in a mode without (*resp.* with) a limit on the number of transistors in series. Next, we elaborate these two modes in Sections IV-A and IV-B, respectively, followed by describing two acceleration techniques in Section IV-C.

Algorithm 1: The flow of MiniTNtk.

```
Input: An $n$-input Boolean function $f$, flag *SeLimit* on whether
       the number of transistors in series has a limit, maximum
       number of transistors in series $M$, and maximum iteration
       number $T$.
Output: A transistor network *tntk* with the minimal number of
        transistors implementing $f$.
1  $r \leftarrow 0$; $tntk \leftarrow NULL$;
2  while True do
3      $SG \leftarrow GenerateBlankSGraph(r)$;
4      $S \leftarrow InitSATProblem(SG)$; $S.addLitSelCons()$;
5      foreach $k$ in $S_{on}$ do
6          $S.addOnsetVars(SG, p_k)$;
7          $S.addOnsetEdgeExistCons()$; $S.addOnPathCons()$;
8          $S.addSourceCons()$; $S.addSinkCons()$;
9          $S.addSimplePathCons()$;
10     foreach $m$ in $S_{off}$ do
11         $S.addOffsetVars(SG, p_m)$;
12         $S.addOffsetEdgeExistCons()$; $S.addSrcSinkCons()$;
13         $S.addNodeSepCons()$;
14     if SeLimit then $S.addOnsetPathLimitCons()$; $t \leftarrow 0$;
15     if $SATSolver(S) = UNSAT$ then $r \leftarrow r + 1$;
16     else
17         $tntk \leftarrow GetTNtk(S.IntnalEdgeVar(), S.LitSelVar())$;
18         if !SeLimit then return tntk;
19         else
20             while True do
21                 if $CheckTranInSeries(tntk, M)$ then return tntk;
22                 $S.addBlockCons()$; $t \leftarrow t + 1$;
23                 if $t > T$ or $SATSolver(S) = UNSAT$ then
24                     $r \leftarrow r + 1$; break;
25                 $tntk \leftarrow GetTNtk(S.IntnalEdgeVar(), S.LitSelVar())$;
```

A. Mode without Limit on Number of Transistors in Series

The input *SeLimit* is set as 0 in this mode. In this mode, Line 1 initializes the number of used transistors r as 0 and *tntk* as *NULL*. Lines 3–13 formulate the generation of a transistor network with r transistors for implementing f as a SAT problem. Line 3 first constructs a blank generalized switching graph *SG* with r switches. Line 4 then sets up a SAT problem instance S and adds all binary variables required for the generalized switching graph construction described in Section III-B. Line 4 also adds a set of literal selection constraints defined in Eq. (1) into S.

Lines 5–9 add the constraints for modeling each onset input pattern into S. Specifically, Line 6 adds the required binary variables for modeling an onset input pattern p_k into S. Lines 7–9 add the constraints defined in Eqs. (2)–(10) into S under the switch configuration of input pattern p_k. Similarly, Lines 10–13 add the constraints for modeling each offset input pattern into S.

After all related constraints are added into S, S is solved by a SAT solver (Line 15). If the result is *UNSAT*, *i.e.*, no satisfying solution can be found, the number of transistors r is increased by 1 (Line 15). Otherwise, Line 17 calls the function *GetTNtk* to construct the corresponding generalized switching graph according to the values of internal edge variables and literal selection variables, and converts it into a transistor network as described in Section III-A.

B. Mode with Limit on Number of Transistors in Series

In this mode, the input *SeLimit* is set as 1. This mode takes two extra inputs, the maximum number of transistors in series, M, and the maximum iteration number, T, which is used for runtime consideration. In this mode, Line 14 adds an extra constraint to limit the number of transistors on the found simple path, *i.e.,* the number of directed switching edges with on-path variables as 1 should be no more than M for each onset input pattern. It is formulated as:

$$\sum_{(i,j)\in E_s} (f_{k,i,j} + f_{k,j,i}) \le M,\ \forall k \in S_{on}. \tag{15}$$

Additionally, Line 14 defines a variable t for recording the number of iterations and sets it as 0.

Although the number of transistors in each found simple path is no more than M, some simple paths containing more than M transistors may still exist in the graph. Lines 20–24 try to restrict the number of transistors in series to be at most M. Specifically, Line 21 checks whether the maximum number of transistors in series is at most M. If so, it returns the derived transistor network; otherwise, Line 22 adds a blocking constraint to exclude the current solution in the next time of solving and increases the iteration number t by 1. If t is larger than the bound T (Line 23), Line 24 directly increases the number of transistors by 1 and continues with the next exact synthesis problem. Otherwise, Line 23 calls the SAT solver to solve the new SAT instance with the blocking constraint added. If the result is *UNSAT* (Line 23), we also go to the next exact synthesis problem (Line 24). Otherwise, Line 25 updates the transistor network, which will be checked for the maximum number of transistors in series again at Line 21.

As a complexity measure, we analyze the number of variables used in our exact synthesis formulation. Clearly, the number of literal selection variables is $O(nr)$. Since the number of all possible internal edges is $O(r^2)$, the number of internal edge variables is $O(r^2)$. The numbers of onset edge existence variables $c_{k,i,j}$ and on-path variables $f_{k,i,j}$ are both $O(2^n r^2)$, since the number of onset input patterns is $O(2^n)$ and the ranges of the subscripts i and j are both $[0, 2r+1]$. Similarly, the number of offset edge existence variables is $O(2^n r^2)$, while that of node separation variables is $O(2^n r)$. Overall, the total number of variables is $O(2^n r^2)$, which is dominated by the onset/offset edge existence variables and on-path variables.

C. Acceleration Techniques

In this section, we propose two acceleration techniques to reduce the solving time of each SAT-based exact synthesis problem. They both first preprocess the given Boolean function f by simplifying its input patterns in S_{on} as an irredundant SOP denoted as $isop_{on}$ and input patterns in S_{off} as an irredundant SOP denoted as $isop_{off}$.

1) Acceleration of Literal Selection: The irredundant SOP $isop_{on}$ consists of a set of different literals, denoted as $L(isop_{on})$. Intuitively, a transistor network can be constructed by using transistors with literals only from the literal set $L(isop_{on})$. Thus, the number of transistors should at least equal the number of literals in $L(isop_{on})$. Based on this, we propose to accelerate MiniTNtk by initializing r in Line 1 of Algorithm 1 as $|L(isop_{on})|$. Besides, we propose to select each switch's literal only from $L(isop_{on})$, and thus, the number of literal selection variables of each switch is reduced to the size of set $L(isop_{on})$, which can result in a further acceleration. Correspondingly, the literal selection constraint shown in Eq. (1) is modified as:

$$\sum_{v\in L(isop_{on})} l_{i,j,v} = 1,\ \forall (i,j) \in E_s. \tag{16}$$

Moreover, for a further acceleration, we also bind each of the first $|L(isop_{on})|$ switches in the generalized switching graph with a unique literal from $L(isop_{on})$. Specifically, if we select a switching edge (i,j) and set its corresponding literal as $L(isop_{on})[v]$, we just add a constraint shown below:

$$l_{i,j,v} = 1. \tag{17}$$

2) Representative Input Patterns: The second acceleration technique exploits unate variables to reduce the number of input patterns needed for modeling the constraint due to the given Boolean function, described in Section III-C. Note that this technique is based on the first acceleration technique. We use an example Boolean function with variable set $\{a, b, c, d\}$, $isop_{on} = (!a \wedge !b \wedge c) \vee (!b \wedge !c \wedge !d) \vee (!a \wedge !d)$ and $isop_{off} = (a \wedge b) \vee (a \wedge c) \vee (b \wedge d) \vee (!c \wedge d)$ to explain the idea. In this example, by definition, variables a, b, d are all negative unate variables of $isop_{on}$. The transistor network is constructed by using transistors with literals from the literal set $\{!a, !b, !c, c, !d\}$ according to the first acceleration technique.

Each product term in $isop_{on}$ (*resp.* $isop_{off}$) represents a set of onset (*resp.* offset) input patterns. For example, the first product term $(!a \wedge !b \wedge c)$ in $isop_{on}$ represents a set of onset input patterns $\{0010, 0011\}$. We only have to model the input pattern 0011 as in Section III-C1 and need not model the input pattern 0010. The reason is that all switching edges bound with literals related to variable d are all bound with $!d$, as variable d is a negative unate variable of $isop_{on}$. Consequently, all these switching edges bound with literals related to variable d do not exist under the pattern 0011. If there exists a simple path in the generalized switching graph under the configuration of the input pattern 0011, then a simple path must also exist under the input pattern 0010, since in this case, more edges exist (*i.e.*, the edges bound with literals $!d$). Thus, the onset input pattern 0010 need not be modeled as long as the onset input pattern 0011 is modeled. We call 0011 an *onset representative input pattern*.

On the other hand, the product term $(a \wedge c)$ in $isop_{off}$ represents offset input patterns $\{1010, 1011, 1110, 1111\}$. We only have to model the input pattern 1010 as in Section III-C2. The reason is that under the configuration of 1010, all switching edges bound with literals related to variables b and d exist, as they are negative unate variables of $isop_{on}$. If the generalized switching graph cannot find a simple path under the configuration of input pattern 1010, then there cannot exist a simple path under the configurations of input patterns $1011, 1110, 1111$, since more switching edges are turned off. We call 1010 an *offset representative input pattern*.

According to the definitions, we derive the onset representative input patterns of each product term in $isop_{on}$ and the offset representative input patterns of each product term in $isop_{off}$ based on unate variables. We model constraints only for all derived onset/offset representative input patterns, leading to acceleration. Note that the use of representative input patterns only reduces the number of input patterns required to be modeled without affecting the optimality.

V. Experimental Results

We implement MiniTNtk in C++. It generates the CNF file representing the SAT-based exact synthesis problem, calls MiniSAT [18] to read the CNF file and solve it, and parses MiniSAT's output file to generate the corresponding transistor network. Since sometimes, such as when a SAT problem is *UNSAT*, the runtime of SAT solving is unpredictable, we set a timeout of 1600s for SAT solving to reduce runtime. In the acceleration techniques proposed in Section IV-C, the irredundant SOPs are obtained by command *simplify* of the logic synthesis tool *sis* [6]. The choice of the maximum number of transistors M in series is same as that of [5], which equals the maximum number of literals in a single product term of the derived irredundant SOP $isop_{on}$. The maximum iteration number T in Algorithm 1 is set as 50. All the following experiments are conducted on a 48-core Intel Xeon Gold 6146 processor using a single thread running at 3.2GHz with 62GB RAM.

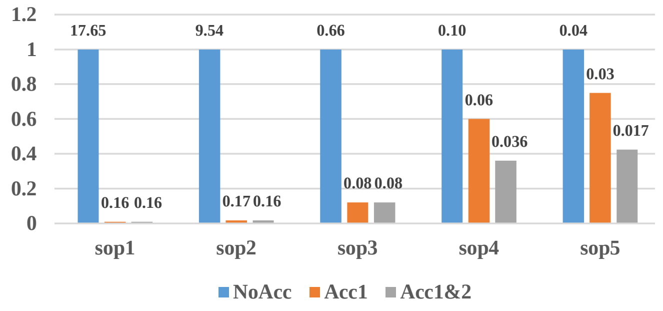

Fig. 4. Runtime comparison of applying three combinations of the two acceleration techniques. The unit of runtime is second.

A. Acceleration Techniques

In this section, we show the impact of the two acceleration techniques proposed in Section IV-C over different Boolean functions. We test the acceleration techniques over different exact synthesis problems for generating transistor networks. Moreover, since the solving time of an *UNSAT* problem can be much longer and unpredictable, we select all the test cases in this section as SAT problems with satisfying solutions. Additionally, our second acceleration technique is closely related to unate variables. We use Boolean functions with different numbers of unate variables in their irredundant SOPs to show how the number of unate variables affects the acceleration.

We test over five 4-input P-class representative Boolean functions.[1] They are listed in the first five rows of Table II in an irredundant SOP form derived by *sis*. Moreover, we also test over a complex function *cpx* with 11 variables and 99 literals from Eq. (14) in [5], which is shown in the last row of Table II. Their corresponding numbers of unate variables are also shown in the third column of Table II. Its last column shows the corresponding minimum number of transistors used to construct a transistor network implementing the Boolean function.

TABLE II
Six irredundant SOPs with corresponding numbers of unate variables and numbers of used transistors.

SOP Name	Irredundant SOP	#Unate Vars	#T
*sop*1	$(!b \wedge !c \wedge !d) \vee (!a \wedge b \wedge !c) \vee (!a \wedge !b \wedge c) \vee (c \wedge d) \vee (a \wedge d)$	0	9
*sop*2	$(!a \wedge b \wedge !c) \vee (!a \wedge !b \wedge c) \vee (a \wedge b \wedge c)$ $\vee (c \wedge d) \vee (b \wedge d) \vee (!a \wedge d)$	1	9
*sop*3	$(!a \wedge b \wedge !c) \vee (!a \wedge !b \wedge c) \vee (!c \wedge d) \vee (!b \wedge d) \vee (!a \wedge d)$	2	8
*sop*4	$(!a \wedge !b \wedge c) \vee (!c \wedge d) \vee (!b \wedge d) \vee (!a \wedge d)$	3	7
*sop*5	$(!b \wedge d) \vee (!a \wedge d) \vee (!a \wedge c) \vee (!a \wedge !b)$	4	6
cpx	Eq. (14) in [5], which has 11 variables and 99 literals	11	11

We first test the runtime on three combinations of the two acceleration techniques over the first five SOPs: 1) without using the two techniques; 2) only using the first acceleration technique; 3) using both acceleration techniques. Note that the second technique cannot be used individually as it is based on the first technique. Fig. 4 shows the runtime comparison of the three combinations, denoted as *NoAcc*, *Acc1*, and *Acc1&2*, respectively. The runtimes of *Acc1* and *Acc1&2* are normalized to that of *NoAcc*, and the runtime of each case is also listed in Fig. 4.

From Fig. 4, we can see that the first acceleration technique plays a key role in reducing the runtime. Especially, for the cases *sop*1, *sop*2, and *sop*3, only using the first acceleration technique can

[1]A P-class is a set of Boolean functions equivalent under the input permutation [20].

reduce the runtime by more than 85%, while the second technique almost has no effect on the runtime reduction. The main reason is that the first technique can greatly reduce the complexity of a SAT problem by reducing the number of literal selection variables of each switching edge and determining some switching edges' literals in advance. However, when solving the cases *sop*4 and *sop*5, the second acceleration technique continues to reduce the runtime significantly by more than 40% compared with only using the first technique. The reason is that they have more unate variables than the first three SOPs. Thus, more input patterns can be removed without modeling them in the SAT problem.

Finally, we remark that the runtime of MiniTNtk for the complex function *cpx* is 0.36s using both acceleration techniques, while MiniTNtk without acceleration cannot find a solution within 1600s. This further confirms the usefulness of our acceleration techniques. In the following experiments, we use both acceleration techniques.

B. *Performance Evaluation of MiniTNtk*

In this section, we compare the performance of MiniTNtk with other related works [8, 11, 13, 12, 5] over three benchmark sets: 1) 53 Boolean functions with optimal handcrafted transistor networks [21] (Section V-B1); 2) all 4-input P-class representative functions (Section V-B2); 3) the complex Boolean function *cpx* listed in Table II (Section V-B3).

1) Evaluation over a Set of Handcrafted Transistor Networks: In this experiment, we compare the performances over a set of 53 Boolean functions with known optimal transistor networks derived manually [21]. These transistor networks do not limit the number of transistors in series. Note that in this experiment, only the NMOS pull-down network for a given Boolean function is generated. The comparison results are shown in Table III, which lists the total number of transistors over all the 53 Boolean functions.

TABLE III
TOTAL NUMBER OF TRANSISTORS IN THE NMOS PULL-DOWN NETWORKS FOR THE 53 BOOLEAN FUNCTIONS FROM [21].

	[21] (optimal)	[8]	[11]	[13]	[12]	[5]	Bound [5]	MiniTNtk	Bound MiniTNtk
#T	356	487	503	516	543	359	383	356	379

From Table III, the graph-based methods [12] and [13] have a worse performance than the factorization-based method [8], while the graph-based method [5] achieves a very good performance in that it only needs 3 more transistors for the 53 Boolean functions compared with the optimal solutions [21] without limiting the number of transistors in series. In contrast, MiniTNtk can achieve the optimal solutions, *i.e.*, 356 transistors in total, for implementing the 53 Boolean functions without limiting the number of transistors in series. Moreover, when limiting the number of transistors in series, MiniTNtk can reduce 4 in total transistors compared to method [5] (*i.e.*, Bound [5] versus Bound MiniTNtk). This experiment demonstrates the optimality of MiniTNtk.

2) Evaluation over All 4-input P-class Representative Boolean Functions: In this experiment, we test over all 3984 4-input P-class representative Boolean functions except the constants 0 and 1. We generate both the pull-up and pull-down networks for each Boolean function using MiniTNtk. The total number of transistors for implementing a Boolean function equals the sum of that of the pull-up network, that of the pull-down network, and that of input inverters. The total number of transistors (*#T*) for implementing all the Boolean functions are shown in Table IV for MiniTNtk and the related works. *Bound* for MiniTNtk and [5] in Table IV corresponds to the mode with a limit on the number of transistors in series.

TABLE IV
TOTAL NUMBER OF TRANSISTORS FOR IMPLEMENTING THE 4-INPUT P-CLASS RERPESENTATIVE BOOLEAN FUNCTIONS.

	[8]	[11]	[13]	[12]	[5]	Bound [5]	MiniTNtk	Bound MiniTNtk
#T	102668	103049	96804	97174	95595	96824	93031	94499

From Table IV, we can see that the graph-based methods [13, 12, 5] outperform the factorization-based method [8]. Among these graph-based methods, the method [5] is the best. Compared with [5] without limiting the number of transistors in series, MiniTNtk outperforms it by 2.68%. Compared with [5] with a limit on the number of transistors in series, MiniTNtk outperforms it by 2.40%. Compared with [8], MiniTNtk without limiting the number of transistors in series reduces the total transistor number by 9.39%. According to Section V-B1, method [5] achieves a near-optimal performance. Thus, although MiniTNtk only makes a small improvement over [5], it is actually a significant achievement. The average runtime of MiniTNtk over all the Boolean functions is 264s (*resp.* 471s) without (*resp.* with) a limit on the number of transistors in series, among which 93.8% (*resp.* 87.8%) functions have runtime no more than 100s.

3) Evaluation over a Complex Boolean Function: This section performs a case study on the complex Boolean function *cpx* listed in Table II. The comparison results are shown in Table V. Our proposed MiniTNtk generates the smallest transistor network with only 11 transistors for implementing the complex function among related works, no matter whether a limit on the number of transistors in series is imposed or not, which is the same as method [5]. MiniTNtk and method [5] can reduce the number of transistors by at least 50% compared to methods [8, 11, 13, 12]. The runtime of MiniTNtk without (*resp.* with) a limit on the number of transistors in series is 0.36s (*resp.* 0.47s).

TABLE V
NUMBER OF TRANSISTORS FOR THE FUNCTION *cpx* IN TABLE II.

	[8]	[11]	[13]	[12]	[5]	Bound [5]	MiniTNtk	Bound MiniTNtk
#T	31	31	25	22	11	11	11	11

VI. CONCLUSIONS AND DISCUSSION

In this work, we propose MiniTNtk, an exact synthesis-based approach for minimizing the transistor network of a given Boolean function. We apply the SAT-based exact synthesis into the transistor network minimization, which consists of three main parts: 1) modeling of generalized switching graph construction; 2) modeling of onset input patterns; 3) modeling of offset input patterns. We also propose two techniques for accelerating the solving of each SAT-based exact synthesis problem. According to the experimental results, it outperforms the state-of-the-art methods.

As for limitations, this work has not considered transistor sizing yet. Also, it cannot handle Boolean functions with multiple outputs [22]. Finally, the SAT-based exact synthesis limits the scalability of MiniTNtk. We will tackle these problems in our future work. Despite these limitations, we believe that the formulation and encoding presented in this work can serve as a important foundation for many other problems related to switch network optimization.

References

[1] S. Bampi and R. Reis. Challenges and emerging technologies for System integration beyound the end of the roadmap of nano-CMOS. In *VLSI-SoC*, pages 21–33, 2009.

[2] R. K. Maurya and B. Bhowmick. Review of FinFET devices and perspective on circuit design challenges. *Silicon*, 14(11):5783–5791, 2022.

[3] H. Kessler et al. Standard cell and supergates designs: An electrical comparison on 4-input logic functions. In *ISCAS*, pages 1744–1748, 2022.

[4] M. Dai et al. Multi-functional multi-gate one-transistor process-in-memory electronics with foundry processing and footprint reduction. *Communications Materials*, 3(1):41–48, 2022.

[5] V. N. Possani et al. Graph-based transistor network generation method for supergate design. *TVLSI*, 24(2):692–705, 2016.

[6] E. M. Sentovich et al. Sequential circuit design using synthesis and optimization. In *ICCD*, pages 328–333, 1992.

[7] M. C. Golumbic, A. Mintz, and U. Rotics. An improvement on the complexity of factoring read-once Boolean functions. *Discrete Applied Mathematics*, 156(10):1633–1636, 2008.

[8] M. G. A. Martins et al. Boolean factoring with multi-objective goals. In *ICCD*, pages 229–234, 2010.

[9] M. G. Martins et al. Functional composition paradigm and applications. In *IWLS*, 2012.

[10] J. Zhu and M. Abd-El-Barr. On the optimization of MOS circuits. *TCAS-I*, 40(6):412–422, 1993.

[11] L. S. da Rosa et al. A comparative study of CMOS gates with minimum transistor stacks. In *SBCCI*, pages 93–98, 2007.

[12] D. Kagaris and T. Haniotakis. A methodology for transistor-efficient supergate design. *TVLSI*, 15(4):488–492, 2007.

[13] V. Possani et al. Optimizing transistor networks using a graph-based technique. *Analog Integrated Circuits and Signal Processing*, 73:841–850, 2012.

[14] L. S. da Rosa et al. Switch level optimization of digital CMOS gate networks. In *ISQED*, pages 324–329, 2009.

[15] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *TC*, 100(8):677–691, 1986.

[16] T. Sasao. *Switching Theory for Logic Synthesis*. Springer Science & Business Media, 2012.

[17] W. Haaswijk et al. SAT-based exact synthesis: Encodings, topology families, and parallelism. *TCAD*, 39(4):871–884, 2019.

[18] E. Niklas. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[19] X. Wang and W. Qian. MinAC: Minimal-area approximate compressor design based on exact synthesis for approximate multipliers. In *ISCAS*, pages 677–681, 2022.

[20] M. Soeken et al. Heuristic NPN classification for large functions using AIGs and LEXSAT. In *SAT*, pages 212–227, 2016.

[21] Logics Lab Federal Univ. Rio Grande do Sul. Catalog of 53 handmade optimum switch networks. http://www.inf.ufrgs.br/logics/docman/53_NSP_Catalog.pdf, 2012.

[22] D. Kagaris. MOTO-X: A multiple-output transistor-level synthesis CAD tool. *TCAD*, 35(1):114–127, 2016.