Don't Care Computation and De Morgan Transformation for Threshold Logic Network Optimization

Chia-Chun Lin, Ciao-Syun Lin, You-Hsuen Tsai,
Yung-Chih Chen, *Member, IEEE*, and Chun-Yao Wang, *Member, IEEE*

***Abstract*—Threshold logic has been attracting great attention from researchers due to the rapid development in nanotechnology-based devices. In the state-of-the-art approach to the threshold logic network (TLN) synthesis using don't cares, we observed that not all the computed don't cares contribute to the cost minimization of threshold logic gate (TLG). Therefore, in this work, we focus on computing the don't cares that effectively provide the opportunities for cost minimization. Furthermore, De Morgan's law for TLGs is applied such that global TLN optimization considering the cost and the number of inverters can be achieved. The experimental results show that the proposed approach is capable of obtaining efficiently a smaller cost and fewer inverters for a set of TLN benchmarks.**

***Index Terms*—De Morgan's law, don't cares, logic implication, threshold logic, threshold logic gate (TLG) optimization.**

I. Introduction

LOGIC synthesis plays a crucial role in the VLSI design flow, and it contains optimization engines to improve the quality of design at logic level. Different from Boolean logic networks, the threshold logic network (TLN) is an alternative representation that is used to express a Boolean function. Due to the higher expression ability of a single threshold logic gate (TLG), a TLN usually has a shorter depth and a smaller number of TLGs compared to the conventional Boolean networks. In the past decade, threshold logic has been attracting great attention from the researchers due to the rapid development in nanotechnology-based devices [2], [28]. Thus, logic synthesis for TLNs has become an important research direction recently.

An integer linear programming (ILP)-based algorithm for the TLN synthesis was first proposed in 2005 [28]. This algorithm performed the binate node splitting operation to obtain unate function nodes. Then, an inequality system was derived from a unate function node. Once the inequality system can be solved by an ILP solver, the function node can be represented with a TLG. Otherwise, this node has to perform the unate node splitting operation until it can be represented with a TLG. That work presented a general approach to generate the inequality system for synthesizing a TLN with the assistance of ILP solvers.

The nanotechnology-based devices, such as resonant tunneling diodes (RTDs) [1], [24], memristors [8], [25], and single-electron transistors (SETs) [11], [27], have been well studied and might be the solutions to the TLN implementation. Since the implementation costs of TLNs correspond to the different nanotechnology-based devices, different cost functions in the TLN synthesis have been proposed in the literature. For example, RTDs are suitable to implement TLGs because their current-voltage characteristics can be exploited to represent complex functionalities as compared to conventional CMOS devices. Since the area of RTD devices determines the weights and threshold values of TLGs, the previous works [10], [16] chose as cost function the summation of all the weights and threshold values in the TLN, and proposed a rewiring-based algorithm to minimize the cost of the given TLN. After the wire removal and rectification network construction, the weights and threshold values of some TLGs in the TLN may be changed such that they cannot be canonically represented. That is, two functionally equivalent TLGs do not have the same weights and threshold value. Therefore, the previous work [10] proposed a simplification procedure for a TLG to obtain its canonical form. On the other hand, since the fabrication of large TLNs is quite challenging, the work [3] simply focused on minimizing the gate count of the given TLN instead. This work exploited the don't cares in the TLN and merged two TLGs with different functionalities while keeping the overall functionality of TLN intact. The previous works [12], [13] also evaluated the quality of TLN by the number of TLGs in it. The authors formulated a collapsing operation for TLGs and proposed an analytic approach for fast circuit transformation.

Recently, the state of the art [4] proposed a don't-care-based algorithm for minimizing the cost of TLGs in a given TLN. This algorithm consisted of two parts: first, computing the

Manuscript received February 17, 2021; revised April 16, 2021; accepted May 30, 2021. Date of publication June 8, 2021; date of current version April 21, 2022. This work was supported in part by the Ministry of Science and Technology (MOST) of Taiwan under Grant 108-2218-E-007-061, Grant 109-2221-E-007-082-MY2, Grant 109-2221-E-155-047-MY2, and Grant 109-2224-E-007-005. This article was recommended by Associate Editor R. Drechsler. *(Corresponding author: Chia-Chun Lin.)*

Chia-Chun Lin and Chun-Yao Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: chiachunlin@gapp.nthu.edu.tw).

Ciao-Syun Lin and You-Hsuen Tsai are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan.

Yung-Chih Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan 32003, Taiwan.

Digital Object Identifier 10.1109/TCAD.2021.3087584

satisfiability don't cares (SDCs) and observability don't cares (ODCs) of a TLG, and second, using the computed SDCs and ODCs to simplify the TLG without changing the structure of TLN. Chen *et al.* [4] showed two sets of experimental results of TLN minimization. One is with TLGs that have at most six inputs, while the other is with TLGs that have at most 15 inputs. According to the experimental results in [4], the TLNs with the 6-input fanin number constraint have much lower cost in terms of summation of total weights and threshold values. This is because the TLGs with more inputs lose the opportunities of TLG sharing. This phenomenon also matches the design principle about the look-up table (LUT) size of modern FPGA devices [7]. Therefore, the fanin number constraint of TLGs in a TLN is suggested to be a small number when the summation of weights and threshold values is adopted as the cost function.

We observed that the the work of [4] can be improved in several aspects. First, because the algorithm only focused on the relations between two inputs of a TLG for computing don't cares, this algorithm might miss some critical don't cares that can be used for TLG optimization. Second, the computed don't cares in the TLG do not always lead to a better optimization result. In other words, the computed don't cares might not contribute to the optimization of TLGs inherently. Third, the inequality constraints for the ILP solvers were constructed by a decision tree. However, different variable orders will lead to different inequality constraints. Therefore, the cost of resultant TLG might not be minimal under a given variable order in [4]. Last, the implementation cost of inverters in the TLNs was not considered in the state of the art. The previous work [24] demonstrated that evaluating the cost of TLNs without considering the implementation cost of inverters is inappropriate.

Thus, in this work, we propose an efficient algorithm to minimize the cost of TLNs consisting of TLGs with at most six inputs while considering the number of inverters in the TLN. Note that this fanin number constraint does not affect the scalability of this work since any TLNs can be synthesized with 6-input TLGs. The main contributions of this work are threefold.

1) This article proposes an algorithm computing the useful don't cares that contribute to TLG optimization.
2) Instead of using the ILP-based approach in the state of the art [4], this article adopts the TLG library mapping to efficiently obtain the optimal-cost TLGs in the TLNs.
3) De Morgan's law for threshold logic is applied and the corresponding inverter optimization algorithm is proposed to achieve a better optimization result.

The remainder of this article is organized as follows. Section II introduces the background of the work. Section III presents the TLN optimization algorithm without structural perturbation. Section IV presents the TLN optimization algorithm with structural perturbation. Section V presents the overall flow of the TLN optimization algorithm. Section VI shows the experimental results. Finally, Section VII concludes this work.

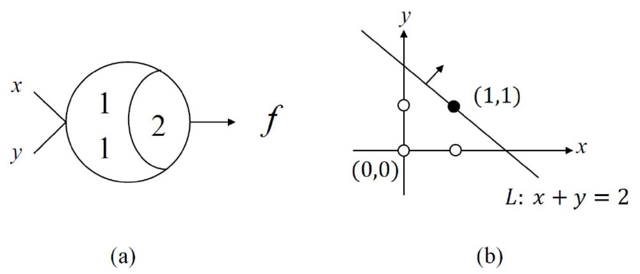


Fig. 1. (a) AND gate. (b) Corresponding hyperplane $L: x + y = 2$.

TABLE I
NUMBER OF n-INPUT NP-TLGS

n	2	3	4	5	6	7	8
NP-TLGs of n variables	2	5	17	92	994	28,262	2,700,791

II. PRELIMINARIES

A. Threshold Logic

A TLG is the primitive element in the TLN. A TLG contains n binary inputs and one binary output. The parameters of a TLG are weights w_i, which correspond to inputs x_i; $i = 1 \sim n$, and a threshold value T. The output f of a TLG is evaluated by (1). If the summation of corresponding weights w_i of inputs x_i that are assumed to be 1 in an input vector is greater than or equal to the threshold value T, the output f is 1. Otherwise, the output f is 0. In the previous works [4], [10], [15], [16], the weights and threshold values are transformed into positive integers for facilitating the cost comparison of TLNs. The TLGs with positive weights and threshold values have the increasing monotonicity property, i.e., $f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \geq f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ for all x_i. In the rest of the article, all the considered TLGs are with positive weights and threshold values

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i w_i \geq T \\ 0, & \text{if } \sum_{i=1}^{n} x_i w_i < T \end{cases} \quad (1)$$

A function that can be represented by a TLG is called a threshold function (TF). There were some studies focusing on the TF identification problem [9], [14], [17], [20], [21], [23]. Table I shows the number of NP-TLGs for up to eight inputs. N and P stand for input negation and permutation, respectively. Chen *et al.* [4] obtained the weights and threshold value of a TLG by using ILP solvers. However, since we only focus on the TLGs that have at most six inputs in this work, the optimal weights and threshold value of 6-input TLGs can be obtained from the 6-input TLG library by permuting the weights for every TLG in the set of NP-TLGs in Table I. The details of this method will be presented in Section III-C.

B. TLG and Hyperplane

A hyperplane is strongly related to a TLG, and it plays an important role in the succeeding discussion. Therefore, in this section, we explain the relationship between the hyperplane and TLG first [14], [15].

According to (1), we know that a TLG has the following parameters $w_1, w_2, \ldots, w_n$ and T. In fact, these parameters compose a hyperplane $H : x_1w_1 + x_2w_2 + \cdots + x_nw_n = T$ in an n-dimensional space. The minterms in the on-set of a TLG will be on or above the hyperplane H. On the other hand, the minterms in the off-set will be below the hyperplane H. For example, the TLG <1, 1; 2> as shown in Fig. 1(a) produces the output of 1 if and only if $(x, y) = (1, 1)$. The behavior of this TLG can be represented in a two-dimensional plane as shown in Fig. 1(b). In Fig. 1(b), when any point that is on or above the hyperplane $L : x + y = 2$ is applied to the TLG, the output f is 1; otherwise, f is 0.

C. SDCs and ODCs

Node optimization is a common technique used for simplifying networks [5], [6], [26]. Some input combinations to the internal nodes of the network will not occur due to the structure and connection of the network. These input combinations are called SDCs to the nodes, and can be used for node optimization. In addition to SDCs, ODCs are also effective to node optimization. ODCs are the input combinations that mask the changes of internal nodes to be observable at the primary outputs.

D. De Morgan's Law for Threshold Logic

In the 1970s, Muroga proposed De Morgan's law for threshold logic, as shown in Fig. 2 [19]. The proof of De Morgan's law is as follows. According to the definition of TLG in (1), the function of the TLG in the left of Fig. 2 can be represented as

$$x_1w_1 + x_2w_2 + \cdots + x_nw_n \geq T. \tag{2}$$

Similarly, the function of the TLG in the right of Fig. 2 can be represented as

$$\overline{x_1}w_1 + \overline{x_2}w_2 + \cdots + \overline{x_n}w_n < T' = w_1 + w_2 + \cdots + w_n + 1 - T \tag{3}$$

where dots represent a complement operation. Then, (3) can be written as

$$T - 1 < (1 - \overline{x_1})w_1 + (1 - \overline{x_2})w_2 + \cdots + (1 - \overline{x_n})w_n. \tag{4}$$

Since $x_1, x_2, \ldots, x_n$ are Boolean variables, (4) is equivalent to

$$x_1w_1 + x_2w_2 + \cdots + x_nw_n > T - 1. \tag{5}$$

Finally, since the weights and threshold value in a TLG are integers, (5) can be rewritten as

$$x_1w_1 + x_2w_2 + \cdots + x_nw_n \geq T. \tag{6}$$

We can see that (6) is the same as (2), which means that the two TLGs in Fig. 2 are equivalent.

According to De Morgan's law for threshold logic, we can transform a TLG in a TLN into its complemented form and vice versa without changing the TLN's functionality.

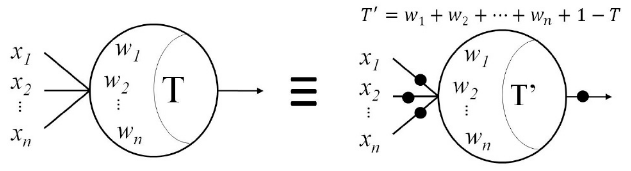


Fig. 2. De Morgan's law for threshold logic.

E. Review of the State of the Art

In this section, we review the previous work [4]. In that work, the authors proposed a don't-care-based minimization algorithm for TLN optimization. The objective of that work is to minimize the summation of weights and threshold values of a TLN without changing its structure. The algorithm aimed to solve two problems: 1) how to compute the don't cares in the TLN and 2) how to use don't cares to simplify a TLG. To solve the first problem, the authors performed logic implications backward and forward on the known assignments. Specifically, for a multiinput TLG g and some of its fanins, f_{in1} and f_{in2}, the known assignments can be obtained by the necessary assignments for g to be observable and an arbitrary value v on the input f_{in1}. Once we can obtain a value w of the input f_{in2} during the logic implication process, the input patterns with $(f_{in1}, f_{in2}) = (v, \overline{w})$ either never occur (i.e., SDC) or make TLG g unobservable (i.e., ODC). Thus, the input patterns with $(f_{in1}, f_{in2}) = (v, \overline{w})$ are don't cares to g.

For the second problem, the authors modeled the TLG optimization problem as an ILP problem. The modeling process derived the greater-than-or-equal-to constraints as well as the less-than constraints. These constraints were obtained by a searching algorithm on a decision tree under a given variable order. In that work, the authors determined the variable order of the decision tree based on the magnitudes of weights in the descending order. These constraints formed an ILP formulation with the objective function of minimizing the summation of weights and threshold value. The solution returned by the ILP solver consisted of the weights and threshold value of the TLG.

III. TLN Minimization Without Structural Perturbation

In this work, the term of structural perturbation is defined as adding or removing inverters, without changing the connectivity of other TLGs in the TLN. In this section, we focus on the don't care computation and TLG optimization without the structural perturbation.

A. ONCVs and OFFCVs

Chen *et al.* [4] computed the SDCs and ODCs by performing logic implications on the TLN. However, this approach might miss the critical don't cares for TLG optimization. For example, Fig. 3 shows a TLN to be optimized. In this graph, the dot marked on an edge is an inverter. Let us consider the g_4 gate, which has three inputs a, b, and g_3. Since for g_1 gate, $a = 1$ and $b = 1$ imply $g_1 = 1$, then $g_1 = 1$ implies $g_3 = 1$ for g_3 gate, the input pattern $(a, b, g_3) = (1, 1, 0)$ to g_4 will never

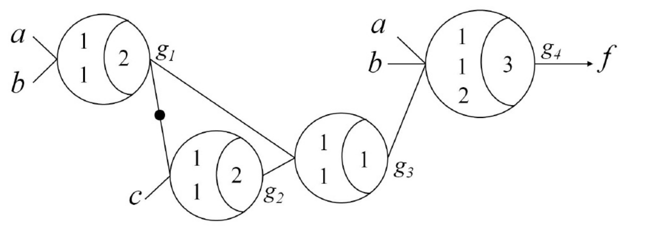


Fig. 3. Example demonstrating the TLG g_4 can be simplified as <1, 1, 1; 2>.

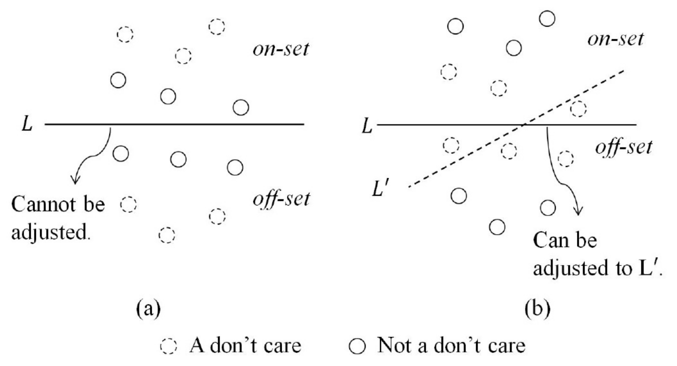


Fig. 4. (a) Hyperplane L that cannot be adjusted. (b) Hyperplane L that can be adjusted to another hyperplane L'.

occur. In other words, the pattern (1, 1, 0) is a don't care to g_4 gate. Thus, the TLG g_4 <1, 1, 2; 3> can be simplified as <1, 1, 1; 2>. However, the technique in [4] cannot compute this don't care since they only assigned *one* input of the TLG during the logic implication process. In this example, both a and b have to be assigned to 1 simultaneously to find out this don't care.

Furthermore, some don't cares cannot contribute to the optimization of TLG. That is, computing certain don't cares is a redundant operation without reducing the cost of TLGs. As mentioned in Section II-B, we know that there exists a hyperplane that separates the on-set and off-set of the TLG. Once the on-set and off-set of TLG are changed, the corresponding hyperplane will be changed accordingly. In the state of the art, some don't cares were computed to optimize the TLG since a don't care can be considered as either an on-set minterm or an off-set minterm. However, as shown in Fig. 4(a), the hyperplane cannot be adjusted for TLG optimization if the computed don't cares are not located on the boundary of the hyperplane. When the don't cares are located on the boundary of the hyperplane, the hyperplane can be adjusted, and a better hyperplane with respect to a lower cost TLG might be obtained. Fig. 4(b) illustrates the idea of hyperplane adjustment for TLG optimization.

In this work, we focus on computing the don't cares that provide the opportunities for optimization. In other words, we only examine if the minterms on the boundary are don't cares or not. Once a minterm on the boundary has been confirmed as a don't care, the boundary will be updated for the succeeding optimization. Before introducing the proposed approach, we first define the *ON critical vectors* (*ONCVs*) and *OFF critical vectors* (*OFFCVs*) of an increasing monotonic function.

Definition 1: Given an input vector $v \in$ on-set (off-set) of an increasing monotonic function, v is said to be an ONCV (OFFCV) if and only if when any bit of v flips from 1 to 0 (0 to 1), the output of the increasing monotonic function also flips from 1 to 0, or 1 to x (0 to 1, or 0 to x), where x denotes a don't care.

By the definitions of ONCVs and OFFCVs, the ONCVs and OFFCVs are the closest minterms to the hyperplane in an n-dimensional space because when any single bit of these minterms flips from 1 to 0 (0 to 1), and the output also flips from 1 to 0, or 1 to x (0 to 1, or 0 to x). Next, we prove the existence of an ONCV and an OFFCV in an increasing monotonic function.

Theorem 1: There exists one ONCV in a nonconstant increasing monotonic function f.

Proof: Let Θ_i^n denote a set of n-input vectors, where the number of 1 in each input vector is i. For example, Θ_1^4 contains four input vectors 0001, 0010, 0100, and 1000.

In this proof, we call the set Θ_i^n a *zero set* if and only if each vector in this set is a don't care or has the output of 0. Since f is a nonconstant increasing monotonic function, we have $f(\Theta_n^n) = 1$ and $f(\Theta_0^n) = 0$. This is because if $f(\Theta_n^n) = 0$, then f is a constant 0 function; and if $f(\Theta_0^n) = 1$, then f is a constant 1 function due to the increasing monotonicity property.

Since $f(\Theta_0^n) = 0$, we obtain a zero set Θ_0^n. The next step is to find an input vector $u \in \Theta_1^n$ such that $f(u) = 1$. If we can have such an input vector u, u is an ONCV by Definition 1. If we cannot find such an input vector, the vector set Θ_1^n is also a zero set. Therefore, we search the ONCVs from the vector set Θ_2^n. By repeating this step, if we cannot find the ONCVs until reaching the vector set Θ_{n-1}^n, we will know that Θ_n^n contains the only ONCV because of $f(\Theta_n^n) = 1$. Thus, there exists one ONCV in a nonconstant increasing monotonic function f. ■

Theorem 2: There exists one OFFCV in a nonconstant increasing monotonic function f.

Proof: Theorem 2 can be proved in a similar way as Theorem 1. Thus, this proof is omitted. ■

For example, a 2-input OR function is a nonconstant increasing monotonic function. The corresponding ONCVs are (1, 0) and (0, 1), and its OFFCV is (0, 0).

B. Update of ONCVs and OFFCVs Considering Don't Cares

Since we want to know whether the minterms on the boundary of hyperplane, i.e., ONCVs and OFFCVs, are don't cares or not, we adopt the method proposed in [4] and [16] to obtain ONCVs and OFFCVs of a TLG. This method is summarized as follows. First, the weights in the TLG are sorted in a descending order. Then, we assign an input variable to be 1 and 0 iteratively. When the variable is assigned to be 1, we check whether the variables that have been assigned can directly determine the output as 1 without considering other nonassigned variables. If the answer is yes, we set the nonassigned variables as 0 and obtain an ONCV. The procedure can be terminated immediately if the assigned values directly determine the output as 0. Fig. 5 is an example showing the process of finding ONCVs of TLG <2, 1, 1; 3>. The OFFCVs can be derived in a similar way. When the variable

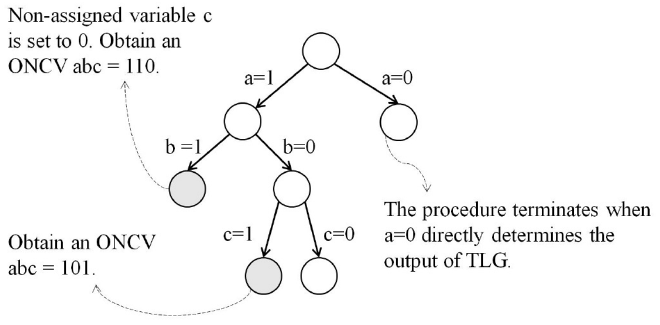

Non-assigned variable c is set to 0. Obtain an ONCV abc = 110.
a=1
a=0
b=1
b=0
c=1
c=0
Obtain an ONCV abc = 101.
The procedure terminates when a=0 directly determines the output of TLG.

Fig. 5. Example showing the procedure of finding ONCVs of TLG <2, 1, 1; 3>.

is assigned to be 0, we check whether the variables that have been assigned can directly determine the output as 0 without considering other nonassigned variables. If the answer is yes, we set the nonassigned variables as 1 and obtain an OFFCV. The procedure can be terminated immediately if the assigned values directly determine the output as 1.

After having the ONCVs and OFFCVs, we next examine if each vector v of them is a don't care of a TLG g or not by using

$$IMP(v \cup OBS(g)) \tag{7}$$

where $OBS(g)$ denotes the set of value assignments that are necessary for a TLG g to be observable at a primary output, and $IMP(V)$ denotes the set of value assignments that are logically implied by a set of known assignments V. Once a conflict occurs during the logic implication process, i.e., a variable in the TLN is mandatorily assigned to 1 and 0 at the same time, the input vector v is a don't care to the TLG g. We use Fig. 3 as an example to explain (7). First, we examine whether $v = (0, 1)$ is a don't care to TLG g_2. Second, the first input of g_3, which is g_1, is set to be $OBS(g_2) = 0$ for propagating g_2 to the primary output. Then, we perform logic implication with these values, which is represented as $IMP(v \cup OBS(g_2))$, on the network in Fig. 3. Since we observe that g_1 is mandatorily assigned to 1 and 0 simultaneously during the logic implication process, the input vector $v = (0, 1)$ is a don't care to the TLG g_2.

If a vector v in the ONCVs or OFFCVs is a don't care to the TLG g, v is not an ONCV or OFFCV anymore. Therefore, we need to update the sets of ONCVs or OFFCVs such that the boundary of hyperplane can be appropriately adjusted. Before introducing the algorithm of ONCVs and OFFCVs update, we define the predecessor and successor of an input vector.

Definition 2: Given an input vector v, a vector set $Predecessor(v)$ of v contains vectors u if and only if u turns into v by flipping one of its input bits from 1 to 0.

Definition 3: Given an input vector v, a vector set $Successor(v)$ of v contains vectors u if and only if u turns into v by flipping one of its input bits from 0 to 1.

For example, Predecessor(000) $= \{100, 010, 001\}$; and Successor(101) $= \{100, 001\}$.

The pseudocode of the algorithm of ONCVs and OFFCVs update is shown in Algorithm 1. In this algorithm, we separately check the boundary of on-set and off-set. At the beginning, the ONCVs and OFFCVs are located on the boundary of on-set and off-set, respectively. However, we will update the ONCVs and OFFCVs when a don't care is found in the ONCVs and OFFCVs. The algorithm is outlined as follows. First, all the original ONCVs are pushed into a queue. Second, we pop a vector v from the queue and examine whether it is a don't care or not. If v is a don't care, we collect it and search its predecessors. A predecessor will be added into the queue when it satisfies the definition of ONCV in Definition 1. Similarly, we next examine if any OFFCV is a don't care. If an OFFCV is a don't care, we collect it and search its successors using the same method. At the end of algorithm, the ONCVs and OFFCVs are updated and the collected don't cares are returned. Note that we only collect the don't cares that provide opportunities for TLG optimization. Thus, this information about don't cares will be used for mapping the minimal-cost TLGs in the TLN.

Algorithm 1: Pseudocode of Algorithm of ONCVs and OFFCVs Update

```
1  Push all the ONCVs into cv_queue
2  while cv_queue is not empty
3      v = Pop a vector from cv_queue
4      if v is a don't care
5          Push v into dc_queue
6          for each Predecessor(v) p
7              Assign cv_flag as true
8              for each Successor(p) s
9                  if the TLG outputs 1 under the pattern s
10                     Assign cv_flag as false
11                     Break the for loop
12             if cv_flag is true
13                 Push p into cv_queue
14 Push all the OFFCVs into cv_queue
15 while cv_queue is not empty
16     v = Pop a vector from cv_queue
17     if v is a don't care
18         Push v into dc_queue
19         for each Successor(v) s
20             Assign cv_flag as true
21             for each Predecessor(s) p
22                 if the TLG outputs 0 under the pattern p
23                     Assign cv_flag as false
24                     Break the for loop
25             if cv_flag is true
26                 Push s into cv_queue
27 return dc_queue
```

According to Definition 1, the ONCVs and OFFCVs determine the hyperplane of a TLG. When some ONCVs or OFFCVs of a TLG are don't cares, we need to update them. After finishing Algorithm 1, the predecessors of ONCVs and the successors of OFFCVs cannot provide the opportunity for TLG optimization. Next we prove the correctness of this claim. Without loss of generality, given a TLG $g = <w_1, \ldots, w_n; T>$, consider one of its ONCV $v = (x_1, \ldots, x_n)$, and one of Predecessor(v) $= u = (y_1, \ldots, y_n)$, where $w_1, \ldots, w_n$ and T are positive integers. According to Definition 1, g outputs 1 under an ONCV. Therefore, we obtain the following inequality:

$$x_1 w_1 + \cdots + x_n w_n >= T. \tag{8}$$

According to Definition 2 about a *Predecessor*(v), we know that there exists $x_i = 0$ and u can be represented as $(x_1, \ldots, x_i + 1, \ldots, x_n)$. Therefore, the output of g under u can be determined by the following inequality:

$$y_1 w_1 + \cdots + y_i w_i + \cdots + x_n w_n = x_1 w_1 + \cdots + (x_i + 1) w_i + \cdots + x_n w_n >= T \tag{9}$$

i.e., the output of g under u must be 1. Thus, u is not an ONCV based on Definition 1. As a result, the predecessors of ONCVs cannot provide the opportunity for TLG optimization. The claim for the successors of OFFCVs can be proved in a similar way. Thus, we omit it in the proof.

For the complexity analysis, given a TLN consisting of n TLGs and the TLGs are with at most k inputs. The complexity for finding ONCVs and OFFCVs of a TLG is $O(2^k)$. Algorithm 1 first checks whether a critical vector is a don't care by performing logic implication in the TLN. Hence, the complexity becomes $O(2^k n)$. If the critical vector is a don't care, the algorithm searches the corresponding successors and predecessors. The maximum numbers of successors and predecessors are both k. Thus, the complexity of optimizing a TLG is $O(2^k nk^2)$. The complexity of optimizing a TLN is $O(2^k n^2 k^2)$. It seems that this complexity is high based on our analysis. However, the algorithm is practically efficient from the implementation viewpoint since this work only focuses on the TLGs with at most six inputs.

We use the TLG g_2 in Fig. 3 as an example to demonstrate Algorithm 1. g_2 has two ONCVs, (0, 1) and (1, 0), and one OFFCV, (0, 0). First, we examine whether the ONCV (0, 1) is a don't care to g_2 by (7). Since (0, 1) is a don't care to g_2, we search its predecessor, which is (1, 1). Since no successor of (1, 1) outputs 1 for TLG g_2, the vector (1, 1) will be pushed into cv_queue for don't care checking. Next, this algorithm iteratively examines the other vectors in cv_queue for don't care checking. Finally, the algorithm returns the dc_queue, which contains two vectors (0, 1) and (0, 0) for further optimization on g_2.

C. Mapping Algorithm Using the TLG Library

Now, we have the truth table of a TLG with don't cares. We are next to find an optimal TLG, in terms of summation of weights and threshold values, which is functionally compatible with the incompletely specified truth table. Although the number of NP-TLGs in Table I is only 994 for 6-input TLGs, we need to consider the permutation of weights in a TLG. For example, two TLGs <1, 2, 1; 3> and <1, 1, 2; 3> can be obtained by permuting the weights from the TLG <2, 1, 1; 3>. In this work, the TLGs with different weight permutations form an isomorphic group (IG). The TLGs in the same IG share two common properties. First, they have the same summation of weights and threshold value. Second, their sizes of on-sets and off-sets are the same. These two properties accelerate the TLG mapping process by pruning a wide range of TLG searching space.

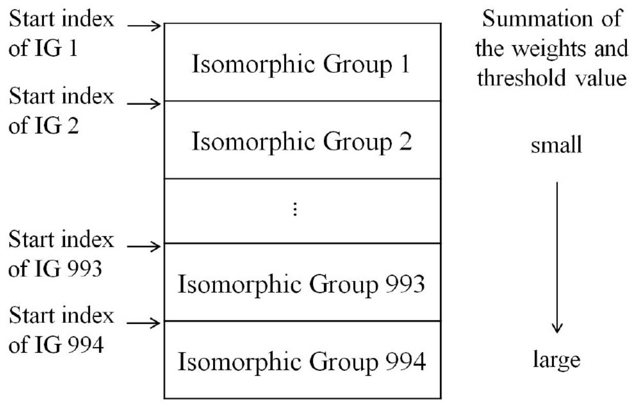


Fig. 6. Searching process for a 6-input compatible TLG. 6-input TLGs have 994 isomorphic groups in total.

Fig. 6 shows how we search the compatible TLG from the TLG library. We first sort the IGs based on the summation of weights and threshold value in the ascending order. Then, we iteratively examine the TLGs. Once we have found a compatible TLG, the algorithm will be terminated immediately because this TLG has the smallest cost. Since the TLGs in this work have at most six inputs, the truth table of a TLG can be stored in a $2^6 = 64$-bit data type. In other words, we can compare the target function and the function of TLGs in the TLG library by using a bitwise-XOR operation, i.e., we XOR the target function and the function of TLG in the library. Once the XORed result returns 1 in the non-don't-care bit position of the target function, the TLG is noncompatible with the target function; otherwise, it is a compatible TLG and has the minimal cost.

Note that we have mentioned that the TLGs in the same IG have the same size of on-sets and off-sets. Given the target function with m_1 minterms in the on-set and m_2 minterms in the off-set, the searching process can directly jump to the next IG if the size of on-set (m_3) of the IG is smaller than m_1 or the size of off-set (m_4) of the IG is smaller than m_2. This is because a TLG in an IG with m_3 on-set minterms and m_4 off-set minterms cannot be compatible to the target function, where $m_3 < m_1$ or $m_4 < m_2$.

IV. TLN Minimization With Structural Perturbation

In Section III, we have proposed the don't care computation and the corresponding TLG optimization algorithm. Although this approach guarantees the minimal-cost TLGs after optimization, we found that [4] resulted in a lower cost for the whole TLN for few benchmarks. This is because optimizing a certain TLG might change the SDCs or ODCs of the other TLGs and affect results. In summary, since we do not change the structure of TLN, the approach in Section III is only capable of generating the minimal-cost

TLGs locally instead of having the minimal-cost TLNs globally. Therefore, in this section, we propose our approach combining De Morgan's law, which allows structural perturbation, for achieving the global TLN optimization.

A. TLN Optimization With De Morgan's Law

According to De Morgan's law of threshold logic, we can consider to replace a TLG with its complemented form in the TLN when the cost is reduced. In fact, since the weights in a TLG and its complemented form are not changed after applying De Morgan's law, we can only compare the threshold values T and T', as shown in Fig. 2. Once a TLG is replaced, the corresponding inverters will be added to the TLN. Note that the total number of inverters in a TLN does not always increase after the replacement. This is because inverter pairs will be removed after the replacement. After examining all the TLGs in the TLN and applying necessary replacements, an optimal-cost TLN is obtained. Although the overall TLN cost might be further reduced after performing other optimizations, e.g., structural rewriting [18], this work only focuses on a lightweight replacement without involving dramatic structural change.

B. Inverter Optimization

In Section I, we use RTDs as an example to explain that the implementation cost of inverters cannot be ignored. However, the implementation cost difference between inverters and the summation of weights and threshold values cannot be evaluated when the device for TLN implementation is not determined yet. Therefore, instead of using a single factor, we evaluate a TLN by considering the number of inverters and summation of weights and threshold values at the same time. That is, we set a balance parameter about the number of inverters and the summation of weights and threshold values to evaluate the quality of the synthesized TLN based on the selected hardware devices.

In the inverter optimization stage, our objective is to reduce the number of inverters in the TLN. We observed that some TLGs, for example, TLG <1, 1, 1; 2>, have the same weights and threshold values as the corresponding complemented forms. Therefore, we can apply De Morgan's law on these TLGs to reduce the number of inverters between two connected TLGs without increasing the cost of TLN In addition to these TLGs, we can also apply De Morgan's law on the other TLGs that have different representations as their complemented forms. In this work, we propose a balance parameter as listed in (10) to determine the TLG that has to be transformed by De Morgan's law again for reducing the number of inverters in the TLN

$$\frac{|\text{Reduced Inverter}|}{\text{Increased Cost}} > \text{User-specified Threshold} \qquad (10)$$

where $|Reduced\ Inverter|$ is the number of reduced inverters and *Increased Cost* is the increased cost when a TLG is transformed to its complemented form. The meaning of (10) is to evaluate the number of inverters that can be reduced per unit cost increment. According to (10), we can obtain a TLN that considers the cost and the number of inverters simultaneously.

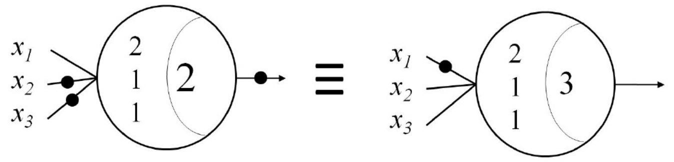


Fig. 7. Example of applying De Morgan's law on TLG <2, 1, 1; 2>.

TABLE II
CPU TIME COMPARISON BETWEEN THE PROPOSED ALGORITHM AND THE BRUTE-FORCE APPROACH

Benchmark	Ours Time (s)	Brute-force Time (s)	Speedup
t481	0.11	0.17	1.55
C1908	0.17	0.34	2.00
rot	0.12	0.30	2.50
C1355	0.15	0.33	2.20
pci_spoci_ctrl	0.36	0.87	2.42
x3	0.13	0.27	2.08
alu4	0.50	1.21	2.42
sasc	0.10	0.23	2.30
simple_spi	0.13	0.29	2.23
dalu	0.57	1.93	3.39
k2	0.73	3.27	4.48
pair	0.24	0.46	1.92
s9234	0.23	0.40	1.74
C5315	0.21	0.36	1.71
C7552	0.42	0.88	2.10
i10	0.68	1.89	2.78
s13207	0.64	1.39	2.17
systemcdes	2.09	8.52	4.08
spi	1.42	3.68	2.59
C6288	0.86	2.46	2.86
des	1.08	4.06	3.76
des_area	7.41	31.32	4.23
tv80	8.00	29.14	3.64
mem_ctrl	4.25	12.24	2.88
systemcaes	6.57	19.35	2.95
usb_funct	5.29	16.87	3.19
ac97_ctrl	2.49	7.56	3.04
aes_core	18.36	53.70	2.92
pci_bridge32	15.80	62.15	3.93
wb_conmax	21.24	70.54	3.32
Average			2.78

For example, Fig. 7 shows a TLG <2, 1, 1; 2> and the corresponding TLG <2, 1, 1; 3> obtained by De Morgan's law. In this example, | *Reduced Inverter* | is 2 and *Increased Cost* is 1. Therefore, the TLG will be replaced by its complemented form when (|Reduced Inverter|/Increased Cost) is greater than a user-specified threshold, say 1.5.

V. Overall Flow

The proposed approach to TLN optimization is shown in Fig. 8. The input is a TLN to be optimized. First, we iteratively select a TLG and compute its ONCVs and OFFCVs. Second, we examine whether the ONCVs and OFFCVs are don't cares. Once an ONCV or OFFCV has been confirmed as a don't care, we update the ONCVs or OFFCVs by examining the corresponding predecessors and successors, respectively, for further optimization. Once all the ONCVs and OFFCVs are not SDCs or ODCs, we search a minimal-cost compatible TLG by bitwise-XOR operations from the TLG library for replacement. When all the TLGs in the TLN have been

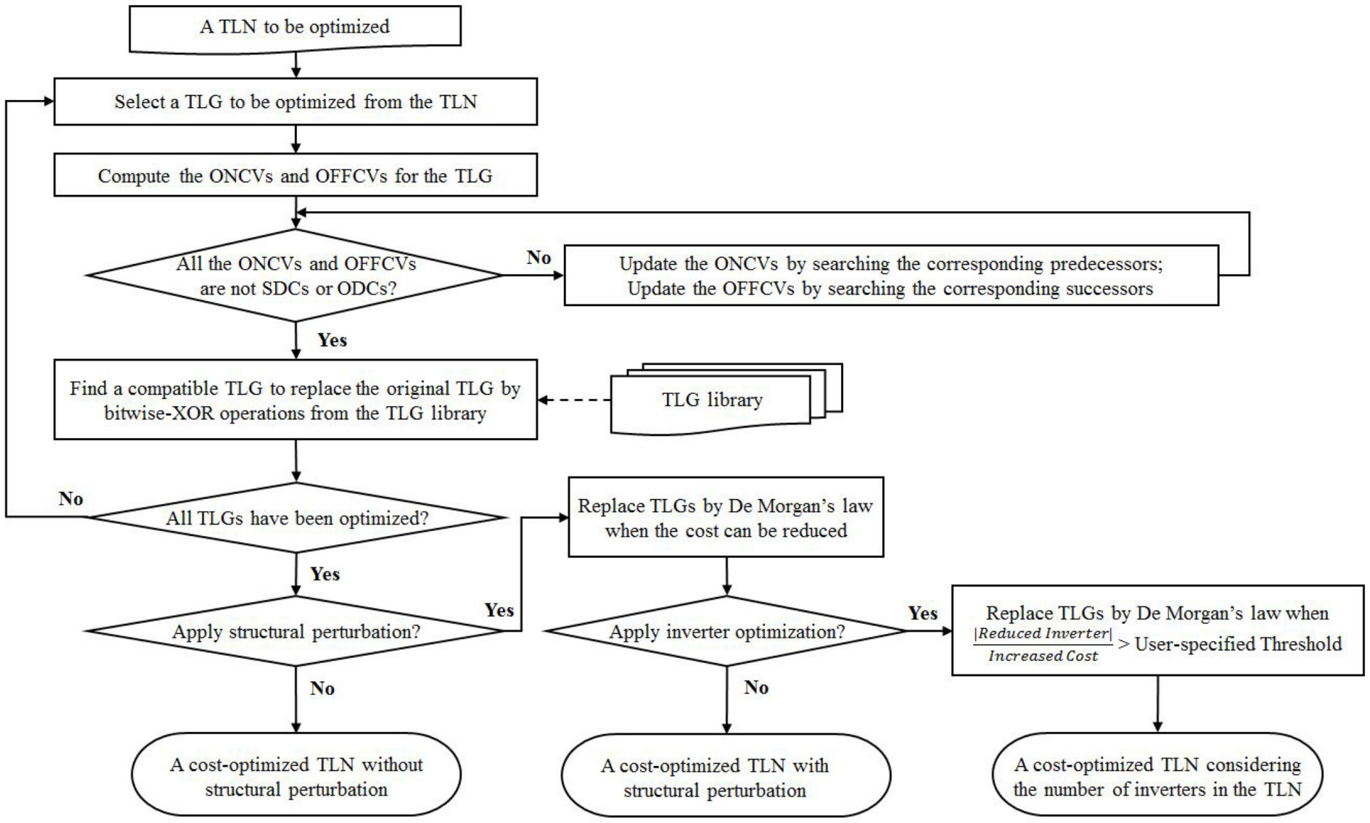

A TLN to be optimized
Select a TLG to be optimized from the TLN
Compute the ONCVs and OFFCVs for the TLG
All the ONCVs and OFFCVs are not SDCs or ODCs?
No
Update the ONCVs by searching the corresponding predecessors; Update the OFFCVs by searching the corresponding successors
Yes
Find a compatible TLG to replace the original TLG by bitwise-XOR operations from the TLG library
TLG library
No
All TLGs have been optimized?
Yes
Replace TLGs by De Morgan's law when the cost can be reduced
Apply structural perturbation?
Yes
Apply inverter optimization?
Yes
Replace TLGs by De Morgan's law when |Reduced Inverter| / Increased Cost > User-specified Threshold
No
No
A cost-optimized TLN without structural perturbation
A cost-optimized TLN with structural perturbation
A cost-optimized TLN considering the number of inverters in the TLN

Fig. 8. Overall flow of the proposed approach to TLN optimization.

processed, a cost-optimized TLN without structural perturbation is obtained. To achieve a better optimization result, the TLGs in the TLN are considered to be transformed by De Morgan's law. When all the TLGs in the TLN have been examined, a cost-optimized TLN with structural perturbation is obtained. Finally, the number of inverters in the TLN can be further reduced with cost overhead when we apply De Morgan's law again on the TLGs that satisfy (8). In the end, a cost-optimized TLN considering the number of inverters is obtained.

VI. Experimental Results

We implemented our approach in C language. The experiments were conducted on a 2.6-GHz Linux platform (CentOS 6.10) with 64-GBytes memory. To have a fair comparison on performance, the experiments for [4] were conducted again on our machine. The source codes were provided by Chen *et al.* [4]. The benchmarks used in the experiments are IWLS 2005 and were provided by [4], which are available online [30]. In this work, we focus on minimizing the cost of TLNs consisting of TLGs with at most six inputs. The TLG library in this experiment is generated by permuting the weights of all 2 to 6-input NP-TLGs [9], [17], [20], [23]. We have verified the optimized TLNs by the combinational equivalence checker *cec* in the ABC package [29]. All the optimization results are correct.

To show the efficiency of the proposed algorithm, we implemented a brute-force checking approach for comparison. That is, the brute-force process checks (7) on all the 2^n vectors to detect don't cares for an n-input TLG. Table II shows the comparison results. Column 1 lists the benchmarks. Columns 2 and 3 show the required CPU time of the proposed algorithm and the brute-force approach, respectively. The last column shows the speedup of our algorithm as compared to the brute-force approach. According to the experimental results, the average speedup of our algorithm is 2.78. In summary, the proposed algorithm is more efficient because it only computes the useful don't cares that contribute to TLG optimization.

Next, we summarize the comparison of experimental results among the previous works [4], [22], and the proposed approach w/wo structural perturbation in Table III. In Table III, Column 1 lists the benchmarks. Columns 2 and 3 show the cost in terms of summation of weights and threshold values in the TLN, and the number of inverters in the TLN by the previous work [22]. Columns 4–7 show the cost, the corresponding cost reduction compared with [22], the number of inverters, and the required CPU time measured in second by [4]. Columns 8–11 show the corresponding results of our approach without structural perturbation. Columns 12–15 show the corresponding results of our approach with structural perturbation.

According to Table III, [4] achieved an average of 10.73% cost reduction, and the required CPU time is 15.48 s on average. Our approach without structural perturbation achieved 12.22% cost reduction and only spent 3.34 s on

TABLE III
COMPARISON OF TLN REDUCTION AND CPU TIME AMONG [22], THE STATE OF THE ART [4], AND THE PROPOSED ALGORITHM

							Ours							
	[22]		[4]				Without structural perturbation				With structural perturbation			
Benchmark	Cost	|Inv|	Cost	R(%)	|Inv|	T(s)	Cost	R(%)	|Inv|	T(s)	Cost	R(%)	|Inv|	T(s)
t481	795	127	678	14.72	124	1.07	673	15.35	127	0.11	532	33.08	135	0.12
C1908	1998	359	1746	12.61	359	1.55	1651	17.37	359	0.17	1397	30.08	353	0.17
rot	2421	357	2320	4.17	354	0.69	2317	4.30	354	0.12	2029	16.19	427	0.13
C1355	2198	422	1926	12.37	422	1.68	1878	14.56	422	0.15	1595	27.43	482	0.16
pci_spoci_ctrl	2914	450	2433	16.51	444	1.92	2459	15.61	450	0.36	2007	31.13	535	0.37
x3	2982	464	2861	4.06	464	0.88	2832	5.03	464	0.13	2481	16.80	529	0.13
alu4	3645	520	2804	23.07	520	3.31	2804	23.07	520	0.50	2351	35.50	605	0.53
sasc	2940	415	2887	1.80	415	0.45	2862	2.65	415	0.10	2579	12.28	637	0.13
simple_spi	3483	491	3299	5.28	491	1.32	3229	7.29	491	0.13	2841	18.43	753	0.15
dalu	4899	757	3775	22.94	744	4.79	3882	20.76	753	0.57	3194	34.80	842	0.75
k2	4262	820	3823	10.30	820	3.04	3835	10.02	820	0.73	2857	32.97	843	0.93
pair	6595	1059	6086	7.72	1059	3.30	5858	11.18	1059	0.24	4936	25.16	1186	0.29
s9234	6236	1032	5621	9.86	1028	3.92	5551	10.98	1032	0.23	4613	26.03	1286	0.27
C5315	6676	1286	5961	10.71	1286	5.48	5753	13.83	1286	0.21	4945	25.93	1328	0.27
C7552	7241	1435	6501	10.22	1435	4.98	6235	13.89	1435	0.42	5347	26.16	1427	0.55
i10	7668	1264	7024	8.40	1264	5.08	6905	9.95	1264	0.68	5754	24.96	1494	0.88
s13207	9245	1453	8444	8.66	1453	5.17	8442	8.69	1453	0.64	7152	22.64	2031	0.83
systemcdes	14744	2624	12163	17.51	2624	14.64	11975	18.78	2624	2.09	9773	33.72	2790	2.65
spi	12861	1978	11834	7.99	1978	9.05	11663	9.31	1978	1.42	10068	21.72	2191	1.86
C6288	14647	2382	12149	17.05	2382	16.10	11470	21.69	2382	0.86	10367	29.22	2932	1.09
des	16556	2281	13698	17.26	2281	19.98	13251	19.96	2281	1.08	11607	29.89	2948	1.42
des_area	24347	4648	19946	18.08	4648	37.30	19568	19.63	4648	7.41	15960	34.45	3327	9.54
tv80	31257	4990	27708	11.35	4986	25.69	27004	13.61	4989	8.00	22541	27.88	5435	10.56
mem_ctrl	29668	5320	27945	5.81	5317	15.40	27635	6.85	5320	4.25	22996	22.49	6327	5.77
systemcaes	39425	7346	35970	8.76	7346	32.56	35258	10.57	7346	6.57	29495	25.19	7912	8.46
usb_funct	56906	10050	52250	8.18	10047	38.92	50933	10.50	10050	5.29	44598	21.63	9305	6.98
ac97_ctrl	54423	9495	52236	4.02	9495	14.58	52095	4.28	9495	2.49	46444	14.66	10748	3.19
aes_core	111739	18783	93523	16.30	18783	103.80	90029	19.43	18783	18.36	73097	34.58	20088	23.67
pci_bridge32	**80657**	**17735**	**77057**	**4.46**	**17735**	**42.93**	**76520**	**5.13**	**17735**	**15.80**	**65332**	**19.00**	**16303**	**17.63**
wb_conmax	142805	27960	140243	1.79	27959	44.73	139409	2.38	27960	21.24	111414	21.98	30916	21.51
Average		4276.77		**10.73**	**4275.43**	15.48		12.22	4276.50	3.34		25.87	4537.17	4.03
Ratio						3.84								1

these benchmarks on average. The proposed approach with structural perturbation achieved 25.87% cost reduction, or 15.14% more as compared with the state of the art, and only spent 4.03 s on the same set of benchmarks on average. Since we did not perform the inverter optimization in the experiment without structural perturbation, the number of inverters is almost the same by our approach and the previous works [4], [22]. The slight difference on the number of inverters is caused by removing some constant nodes after optimization.

The average number of inverters by our approach with structural perturbation is approximately 260, higher than that without structural perturbation. However, the number of inverters with structural perturbation does not always increase. For example, considering the benchmark pci_bridge32, ours with structural perturbation obtained 16 303 inverters while that without structural perturbation obtained 17 735 inverters. In summary, the proposed approach with structural perturbation is very efficient and results in much more cost reduction.

Besides, we also show the experimental results with the inverter optimization under different user-specified threshold parameters in Table IV. Column 1 lists the benchmarks. Columns 2 and 3 show the cost reduction and the number of inverters without applying the inverter optimization. Columns 4–13 show the corresponding results with applying the inverter optimization under user-specified thresholds of 2.0, 1.6, 1.2, 0.8, and 0.4, respectively.

According to (10), the higher user-specified threshold is the more TLGs cannot be replaced for inverter optimization. As a result, the optimized TLN with a high user-specified threshold usually contains more inverters. For example, the average number of inverters is 3256.50 when the user-specified threshold is 2.0, but the average number of inverters is 2121.63 when the user-specified threshold is 0.4.

Since the implementation cost of an inverter depends on the selected hardware device, the user-specified threshold provides the flexibility for the inverter optimization algorithm. That is, we can assign the user-specified threshold as a large number when the implementation cost of inverters is relatively low. Otherwise, we can assign the user-specified threshold as a small number to achieve a lower implementation cost.

To show the advantage of the proposed overall approach with structural perturbation and inverter optimization over [4], we take the results from Table IV under the column of Threshold = 2.0 for comparison. Our result achieved 13.73% (24.46%–10.73%) more cost reduction and 1018.93 (4275.43–3256.5) more inverter reduction on average as compared to the state of the art. The required CPU time for the results of Threshold = 2.0 is 4.44 s on average. Therefore, our approach is 3.49 times faster than the state of the art.

TABLE IV
TLN REDUCTION AND THE NUMBER OF INVERTERS UNDER DIFFERENT INVERTER OPTIMIZATION THRESHOLDS

Benchmark	Without inv. opt.		Threshold = 2.0		Threshold = 1.6		Threshold = 1.2		Threshold = 0.8		Threshold = 0.4	
	R(%)	|Inv|	R(%)	|Inv|	R(%)	|Inv|	R(%)	|Inv|	R(%)	|Inv|	R(%)	|Inv|
t481	33.08	135	32.33	127	32.33	112	32.08	111	30.44	103	27.55	92
C1908	30.08	353	28.83	239	28.53	239	28.33	233	27.73	226	27.33	223
rot	16.19	427	15.32	361	14.58	359	13.80	335	12.93	292	10.37	268
C1355	27.43	482	25.93	322	25.93	322	25.93	322	23.93	283	23.16	333
pci_spoci_ctrl	31.13	535	28.41	368	29.20	363	29.00	349	28.48	332	27.08	314
x3	16.80	529	16.47	498	16.47	498	14.79	423	14.08	400	12.04	369
alu4	35.50	605	34.35	445	34.21	445	33.64	427	32.37	386	31.14	353
sasc	12.28	637	10.71	454	9.76	407	9.56	398	7.48	323	6.67	298
simple_spi	18.43	753	16.57	513	16.57	513	16.31	500	13.75	407	12.89	389
dalu	34.80	842	32.95	529	32.88	524	32.25	477	30.43	376	29.80	355
k2	32.97	843	31.63	687	31.42	670	30.34	609	27.48	461	23.06	315
pair	25.16	1186	23.59	845	23.37	790	22.85	746	20.91	624	19.76	581
s9234	26.03	1286	24.37	892	24.18	873	23.60	827	21.44	686	20.29	638
C5315	25.93	1328	23.70	879	23.65	874	23.37	859	22.32	784	21.67	756
C7552	26.16	1427	24.36	1002	24.28	994	24.02	972	22.57	875	21.76	839
i10	24.96	1494	23.72	1147	23.59	1132	23.00	1072	21.50	963	19.64	891
s13207	22.64	2031	20.92	1517	20.88	1519	20.44	1468	17.69	1183	15.72	1089
systemcdes	33.72	2790	32.22	1811	31.84	1747	31.41	1670	30.07	1483	28.98	1340
spi	21.72	2191	21.14	1740	21.03	1712	20.85	1693	18.40	1371	17.46	1316
C6288	29.22	2932	27.02	1927	26.97	1918	26.24	1772	24.51	1525	23.44	1417
des	29.89	2948	28.36	1971	28.37	1958	28.17	1915	26.52	1688	25.93	1635
des_area	34.45	3327	33.83	2101	33.82	2093	33.73	2069	31.81	1584	31.54	1613
tv80	27.88	5435	26.58	3925	26.48	3886	26.08	3714	24.16	3014	23.19	2809
mem_ctrl	22.49	6327	21.15	4506	21.09	4488	21.00	4455	17.77	3411	17.22	3315
systemcaes	25.19	7912	23.78	5659	23.62	5592	23.16	5332	20.32	4217	19.43	3999
usb_funct	21.63	9305	20.79	7454	20.79	7463	20.61	7325	19.22	6527	18.69	6328
ac97_ctrl	14.66	10748	13.43	8684	13.42	8674	13.28	8558	10.55	7092	10.25	6986
aes_core	34.58	20088	33.26	14986	33.08	14629	32.40	13624	30.70	11601	29.28	10705
pci_bridge32	19.00	16303	18.20	11717	18.19	11645	18.08	11580	15.48	9278	13.92	8590
wb_conmax	21.98	30916	19.89	20389	19.88	20338	19.53	19395	15.03	6013	14.35	5493
Average	25.87	4537.17	**24.46**	**3256.50**	24.35	3225.90	23.93	3107.67	22.00	2250.27	20.79	2121.63

VII. CONCLUSION

In this work, we proposed an optimization algorithm for TLNs. The algorithm searches ONCVs and OFFCVs of a TLG and exploited the obtained don’t cares to map the optimal TLG from the TLG library. Then, De Morgan’s law is applied to the TLGs such that the global-optimal TLN considering the cost and the number of inverters can be achieved. The experimental results show that the proposed approach is capable of obtaining a smaller cost and fewer inverters in a TLN for a set of benchmarks in a more efficient way.

REFERENCES

[1] T. Akeyoshi, K. Maezawa, and T. Mizutani, “Weighted sum threshold logic operation of MOBILE using resonant-tunneling transistors,” *IEEE Electron Device Lett.*, vol. 14, no. 10, pp. 475–477, Oct. 1993.

[2] V. Beiu, J. M. Quintana, and M. J. Avedillo, “VLSI implementations of threshold Logic—A comprehensive survey,” *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1217–1243, Sep. 2003.

[3] Y.-C. Chen, L.-C. Zheng, and F.-L. Wong, “Optimization of threshold logic networks with node merging and wire relacement,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 24, no. 6, p. 67, 2019.

[4] Y.-C. Chen, H.-J. Chang, and L.-C. Zheng, “Don’t-care-based node minimization for threshold logic networks,” in *Proc. DAC*, 2020, pp. 1–6.

[5] M. Damiani and G. De Micheli, “Observability don’t care sets and Boolean relations,”Ì in *ICCAD Tech. Paper*, 1990, pp. 502–505.

[6] M. Damiani and G. De Micheli, “Don’t care set specifications in combinational and synchronous logic circuits,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 3, pp. 365–388, Mar. 1993.

[7] H. Gao, Y. Yang, X. Ma, and G. Dong, “Analysis of the effect of LUT size on FPGA area and delay using theoretical derivations,” in *Proc. Int. Symp. Qual. Electron. Design*, 2005, pp. 370–374.

[8] L. Gao, F. Alibart, and D. B. Strukov, “Programmable CMOS/memristor threshold logic,” *IEEE Trans. Nanotechnol.*, vol. 12, no. 2, pp. 115–119, Mar. 2013.

[9] T. Gowda, S. Vrudhula, and G. Konjevod, “A non-ILP based threshold logic synthesis methodology,” in *Proc. IWLS*, 2007, pp. 222–229.

[10] P.-Y. Kuo, C.-Y. Wang, and C.-Y. Huang, “On rewiring and simplification for canonicity in threshold logic circuits,” in *Proc. ICCAD*, 2011, pp. 396–403.

[11] C. R. Lageweg, S. D. Cotofana, and S. Vassiliadis, “A linear threshold gate implementation in single electron technology,” in *Proc. IEEE Comput. Soc. VLSI Workshop*, 2001, pp. 93–98.

[12] N.-Z. Lee and J.-H. R. Jiang, “Constraint solving for synthesis and verification of threshold logic circuits,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 5, pp. 904–917, May 2021.

[13] N.-Z. Lee, H.-Y. Kuo, Y.-H. Lai, and J.-H. R. Jiang, “Analytic approaches to the collapse operation and equivalence verification of threshold logic circuits,” in *Proc. ICCAD*, 2016, pp. 30–37.

[14] C.-C. Lin, C.-H. Liu, Y.-C. Chen, and C.-Y. Wang, “A new necessary condition for threshold function identification,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5304–5308, Dec. 2020.

[15] C.-C. Lin, C.-W. Huang, C.-Y. Wang, and Y.-C. Chen, “In&Out: Restructuring for threshold logic network optimization,” in *Proc. Int. Symp. Qual. Electron. Design*, 2017, pp. 413–418.

[16] C.-C. Lin, C.-Y. Wang, Y.-C. Chen, and C.-Y. Huang, “Rewiring for threshold logic circuit minimization,” in *Proc. DATE*, 2014, pp. 1–6.

[17] C.-H. Liu, C.-C. Lin, Y.-C. Chen, C.-C. Wu, C.-Y. Wang, and S. Yamashita, “Threshold function identification by redundancy removal and comprehensive weight assignments,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 12, pp. 2284–2297, Dec. 2019.

[18] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis,” in *Proc. DAC*, 2006, pp. 532–535.

[19] S. Muroga, *Threshold Logic and Its Applications*. New York, NY, USA: Wiley, 1971.

[20] A. Neutzling, M. G. A. Martins, V. Callegaro, A. I. Reis, and R. P. Ribas, "A simple and effective heuristic method for threshold logic identification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 5, pp. 1023–1036, May 2018.

[21] A. Neutzling, M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Synthesis of threshold logic gates to nanoelectronics," in *Proc. Symp. Integr. Circuits Syst. Design*, 2013, pp. 1–6.

[22] A. Neutzling, J. M. A. Matos, A. Mishchenko, A. I. Reis, and R. P. Ribas, "Effective logic synthesis for threshold logic circuit design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 926–937, May 2019.

[23] A. K. Palaniswamy and S. Tragoudas, "An efficient heuristic to identify threshold logic functions," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 8, no. 3, pp. 1–17, 2012.

[24] H. Pettenghi, M. J. Avedillo, and J. M. Quintana, "Improved nanopipelined RTD adder using generalized threshold gates," *IEEE Trans. Nanotechnol.*, vol. 10, no. 1, pp. 155–162, Jan. 2011.

[25] J. Rajendran, H. Manem, R. Karri, and G S. Rose, "Memristor based programmable threshold logic array," in *Proc. Nanoarch*, 2010, pp. 5–10.

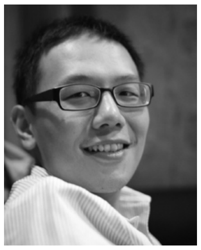
[26] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Proc. DAC*, 1990, pp. 297–301.

[27] M. H. Sulieman, and V. Beiu, "Characterization of a 16-bit threshold logic single electron technology adder," in *Proc. ISCAS*, pp. 681–684, 2004.

[28] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, "Threshold network synthesis and optimization and its application to nanotechnologies," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 107–118, Jan. 2005.

[29] Berkeley Logic Synthesis and Verification Group. (2007). *ABC: A System for Sequential Synthesis and Verification, Release 70930*. [Online]. Available: http://www.eecs.berkeley.edu/∼alanmi/abc/

[30] Y.-C. Chen, H.-J. Chang, and L.-C. Zheng. (2019). *Benchmark Circuits Used in the Work Entitled 'Don't-Care-Based Node Minimization for Threshold Logic Networks'*. [Online]. Available: http://doi.org/10.5281/zenodo.3525945

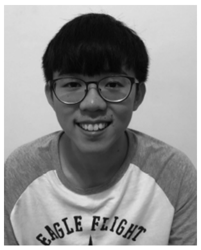


Chia-Chun Lin received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2011, 2013, and 2021, respectively.

His current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.

Ciao-Syun Lin received the B.S. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2020, where she is currently pursuing the M.S. degree.

Her current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.



You-Hsuen Tsai received the B.S. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2020, where he is currently pursuing the M.S. degree.

His current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.

Yung-Chih Chen (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2003, 2005, and 2011, respectively.

He is currently an Associate Professor with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan. His current research interests include logic synthesis, design verification, and design automation for emerging technologies.

Chun-Yao Wang (Member, IEEE) received the Ph.D. degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2002.

Since 2003, he has been an Assistant Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu, where he is currently a Distinguished Professor. He has published over 80 technical papers in these areas and is a named inventor in nine patents. His current research interests include logic synthesis, optimization, and verification for very large-scale integrated/system-on-chip designs and emerging technologies.

Dr. Wang was the recipient of the Best Paper Award in 2018 IEEE International Symposium on VLSI Design, Automation and Test. Two of his research results were nominated as Best Papers in the 2009 IEEE Asia and South Pacific Design Automation Conference and the 2010 IEEE/ACM Design Automation Conference, respectively. In 2020, he was awarded the Distinguished Electrical Engineering Professor Award from the Chinese Institute of Electrical Engineering, Taiwan.