Fast Synthesis of Threshold Logic Networks with Optimization

Yung-Chih Chen, Runyi Wang, and Yan-Ping Chang
Department of Computer Science and Engineering
Yuan Ze University, Taoyuan, Taiwan

***Abstract*—Threshold logic, a more compact Boolean representation compared to conventional logic gate representation, re-attracted substantial attention from researchers due to the advances of threshold logic implementations with novel nanoscale devices. For the compact representation to be promising, a fast and effective method for transforming a conventional Boolean logic network into a threshold logic network is necessary. This paper presents such a synthesis method for threshold logic based on logic optimization. First, a Boolean logic network is mapped into a threshold logic network by one-to-one mapping. Then, a method is used to optimize the threshold logic network based on eight transformations for reducing gate count. Unlike the previous methods, the proposed method does not require threshold function identification, and thus is much more efficient. The experimental results show that the proposed method is three orders of magnitude faster than a widely used synthesis method. Additionally, the proposed method has a better synthesis quality with an average saving of 28% threshold gates.**

I. Introduction

Threshold logic, which is composed of linear threshold gates, is an alternative representation to Boolean logic. The logical function f of a linear threshold gate with n inputs, $x_1 \sim x_n$, is defined as follows:

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

Here, there is a threshold value T and each binary input x_i has a weight w_i. For an input vector, if the sum of the weights of the inputs whose values are 1, is greater than or equal to T, f is 1; otherwise, f is 0. Fig. 1 shows the graphical representation of the linear threshold gate. A Boolean function is called a *threshold function* if it can be expressed with a single threshold gate.

The development of threshold logic started in $1960s$ [15] [16]. Compactness is one of the major advantages of threshold logic. For representing a Boolean function, a threshold logic network (TLN) usually has fewer logic gates compared to the conventional Boolean logic network. For example, Fig. 2(a) shows a Boolean logic network for the function $f = x_1(x_2 + x_3 + (x_4(x_5 + x_6)))$. There are totally four logic gates. However, for threshold logic, only one threshold gate is required for representing f as shown in Fig. 2(b).

Although there were several research achievements on threshold logic in the early days, it had little impact on the conventional IC design due to the lack of effective hardware realization. In the recent decades, the advances on nanoscale devices make threshold logic re-attracted substantial attention from researchers due to its higher compatibility for implementing ICs with the nanoscale devices [1] [3] [11] [17] [19].

Although it might still require much time and effort for realizing threshold logic with the nanoscale devices, we believe that using threshold logic as an intermediate representation in today's electronic design automation flow is worthy of being studied. The compactness characteristic of threshold logic makes it easier to be analyzed and manipulated.

For example, a gate-level design to be verified could be first transformed into threshold logic and then verification is conducted in the threshold logic domain for reducing complexity. Moreover, a design could be first transformed into threshold logic, then optimized in the threshold logic domain, and finally transformed back to the conventional Boolean logic. An optimization method in the threshold logic domain could find some optimization opportunities that are hard to be identified in the conventional Boolean logic domain.

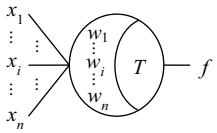

Fig. 1. The graphical representation of a linear threshold gate.

For the above ideas to be promising, an effective and efficient method of transforming a conventional Boolean logic network into a TLN is necessary. Thus, the objective of this paper is to develop a fast and high-quality synthesis method for TLNs.

In fact, there have been several synthesis methods proposed for TLNs. They all work based on a threshold function identification procedure, which is integer linear programming (ILP)-based [1] [14] [18], binary decision diagram (BDD)-based [4] [5] [12], or truth table-based [10] [12].

The ILP-based methods model the problem of checking whether a Boolean function is a threshold function or not as an ILP problem. If the Boolean function is a threshold function, the minimum weights and threshold value of the corresponding threshold gate are derived from the ILP solution. Otherwise, the Boolean function is decomposed into multiple sub-functions to be further processed recursively. Since a potentially large number of ILP solving calls are required, inefficiency is the main issue of the ILP-based methods.

On the other hand, the non-ILP-based methods construct the BDD or truth table of the Boolean function and then determine whether it is a threshold function by analyzing the BDD or truth table. If it is, the weights and threshold value which may not be the minimum are computed. Although there is no computation-intensive ILP solving call, BDD or truth table construction could affect the scalability of the methods.

In addition to the complexity of threshold function identification, these previous methods could have another issue when synthesizing a Boolean logic network. If the given network is composed of only primitive logic nodes, such as AND, OR, and NOT, a process to cluster nodes for forming complex Boolean functions during or before synthesis is required by these methods. Otherwise, the resultant TLN will be same with that obtained by mapping each primitive logic node into a threshold gate, i.e., one-to-one mapping. The gate count remains the same.

The ILP-based method [18] works together with a heuristic for clustering nodes during synthesis. Thus, although the method is exact for threshold function identification, it is not exact for threshold logic synthesis. Our experiments show that the synthesis results obtained by the method can be improved. On the other hand, the works in [4] and [5] used the logic optimization scripts in the SIS package [13] to optimize the Boolean logic network before synthesis, such that the resultant Boolean logic network is composed of nodes with complex functions. However, this pre-process could be non-scalable for circuits of present-day complexity.

In recent years, the and-inverter graph (AIG) is widely used for representing a Boolean logic network [2]. For the TLN to be a promising logic representation, an effective synthesis method should be able to deal well with a Boolean logic network in the AIG format. Thus, in this paper, we also aim to propose such a synthesis method.

We take a different look at the synthesis problem of threshold logic. In fact, AND, OR, and NOT functions are threshold functions. Transforming a Boolean logic network composed only of them into

This work was supported in part by the Ministry of Science and Technology of Taiwan under grant MOST 104-2220-E-155-001.

978-1-4673-9569-4/16/$31.00 ©2016 IEEE

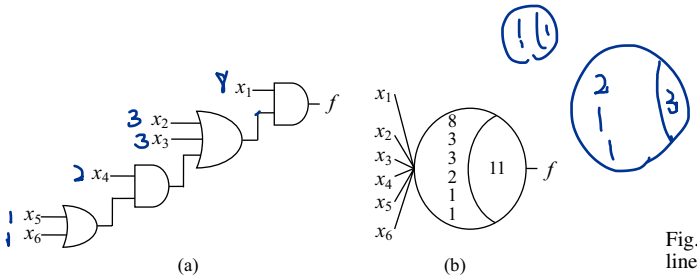

Fig. 2. Representations for a Boolean function $f = x_1(x_2+x_3+(x_4(x_5+x_6)))$: (a) Conventional Boolean logic and (b) threshold logic.

a TLN can be easily achieved by mapping each gate into a threshold gate. After obtaining such a TLN, we can focus on optimizing it to generate a more compact one. Thus, we propose a TLN optimization method for reducing threshold gate count. The method works based on eight logic transformations. Each transformation corresponds to a sufficient condition on gates which can be merged or removed without affecting the overall functionality. The overall optimization process identifies the gates that satisfy the sufficient conditions and performs the transformations. It does not need to repeatedly check whether a Boolean function is a threshold function or not, and therefore, is much more efficient than the previous methods.

We conducted the experiments on a set of benchmarks from the IWLS 2005 benchmark suite, which are expressed in the AIG format. We compared the proposed method with the ILP-based synthesis method [18], which can directly synthesize a Boolean logic network in the AIG format as well. The experimental results show that our method can save an average of 28% threshold gates. Additionally, our method is much more efficient. For every benchmark, our method spent less than one second for completing synthesis. However, the largest benchmark with 88854 gates cost the ILP-based method approximately 3 hours.

The remainder of this paper is organized as follows: Section II reviews some background on threshold logic. Section III shows an illustrative example to demonstrate the intention of the proposed synthesis method. Section IV presents the proposed optimization method. Section V shows the experimental results. Finally, the conclusion is presented in Section VI.

II. Background

As introduced in Section I, a TLN is composed of threshold gates (or nodes). A threshold gate is a multi-input and one-output gate, in which there is a threshold value T and each binary input x_i is companied with a weight w_i. A Boolean function that can be realized by a single threshold gate is called a threshold function.

The traditional primitive logic operations AND, OR, and NOT are threshold functions. A Boolean logic network composed only of them can be easily transformed into a TLN by one-to-one mapping. For example, an n-input AND function can be realized with a threshold gate in which the threshold value is n and all the weights are 1. An n-input OR function can be realized with a threshold gate in which the threshold value and all the weights are 1. The NOT function can be realized with a threshold gate in which the threshold value is 0 and the weight is -1.

A Boolean function $f(x_1, x_2, ..., x_n)$ is said to be positive unate in a variable x_i, if $f_{x_i} \geq f_{\overline{x_i}}$. Likewise, it is negative unate in x_i if $f_{x_i} \leq f_{\overline{x_i}}$. If f is either positive or negative unate in every x_i, f is said to be unate; otherwise, f is binate [6]. Unateness is an important property of a threshold function. Any threshold function is a unate function [6]. For a threshold function, if all the weights are positive values, it is positive unate. If all the weights are negative values, it is negative unate.

In most of the previous studies on threshold logic [4] [5] [10] [12] [14] [18], the weights and the threshold value of a threshold gate are assumed to be positive or negative integers. Additionally, for easy to analyze a threshold gate, all the negative weights can be transformed into positive ones (i.e., a positive unate form). The transformation procedure is as follows [7] [9]: First, one of the negative weights is selected. Then, the negative weight is set to its absolute value and the corresponding input polarity is reversed. After that, the magnitude of the negative weight is added to the threshold value. Finally, the above steps are repeated until all the weights become positive.

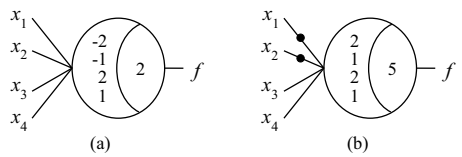

Fig. 3. (a) A linear threshold gate having negative weights. (b) The resultant linear threshold gate by transforming all the negative weights to be positive.

For example, Fig. 3(a) shows a threshold gate having negative weights. By using the above method, we can get a functionally equivalent threshold gate which has only positive weights as shown in Fig. 3(b). Here, we use a dot to indicate that an input has a negative polarity and the dot can be seen as a NOT gate. This example also shows that a NOT gate can be eliminated by integrating it into its fanout gate(s) using the positive weight transformation method in reverse.

In this work, we also make all the weights positive before we optimize a TLN. Additionally, after optimizing a TLN, we integrate all the NOT gates to their fanout gates.

For a threshold gate g in the positive unate form (i.e., all the weights are positive numbers), we say that g has a *controlling input*, if the value of the input can determine the output value of g regardless of the other inputs. A controlling input is either a *controlling*-1 *input* or a *controlling*-0 *input* according to the output value it determines. Suppose that x_i is an input of g with the weight w_i. If w_i is greater than or equal to the threshold value of g, $x_i = 1$ determines $g = 1$ regardless of the other inputs. Then, x_i is said to be a controlling-1 input. Furthermore, if the sum of all the weights except w_i is less than the threshold value of g, $x_i = 0$ determines $g = 0$ regardless of the other inputs. Then, x_i is said to be a controlling-0 input. In Fig. 3(b), x_3 is a controlling-0 input and x_1 is also a controlling-0 input with a negative polarity.

III. An Illustrative Example

In this section, we use an example to demonstrate the intention of the proposed synthesis method. We assume all the Boolean logic networks under consideration consist of only AND, OR and NOT gates. Complex gates can be decomposed into these gates before synthesis.

Let us consider the Boolean logic network in Fig. 4(a). First, we transform it into a TLN by one-to-one mapping as shown in Fig. 4(b). Next, we start to optimize the TLN. We present eight logic transformations for reducing gate count by merging gates, which will be detailed in the next section. Each transformation comes with a sufficient condition. To optimize the TLN, we identify gates that satisfy the sufficient conditions and then perform the transformations. Fig. 4(c) shows the optimized TLN obtained by merging the gates g_1 and g_5, the gates g_2 and g_3, and the gates g_2, g_4, and g_6. Finally, we eliminate the NOT gates by using the positive weight transformation method in reverse, and get the resultant TLN as shown in Fig. 4(d). In the overall synthesis process, we do not need to repeatedly check whether a Boolean function is a threshold function or not.

IV. Threshold Logic Optimization

In this section, we first present the transformations on threshold logic optimization. Two of them are adapted from the previous works [9] [18] and some are derived from Boolean algebra. These transformations work under an important assumption that all the weights of a threshold gate are positive values. Then, we present a simple but effective method for threshold logic optimization based on the transformations.

A. Transformations on threshold logic

Transformation 1: Constant gate elimination

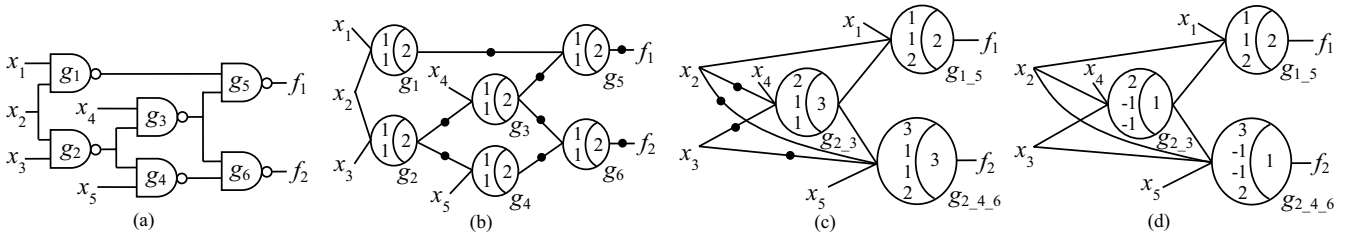

Fig. 4. An example of demonstrating the intention of the proposed synthesis method. (a) The given Boolean logic network. (b) The TLN obtained by one-to-one mapping. (c) The optimized TLN by merging gates. (d) The resultant TLN by eliminating the NOT gates.

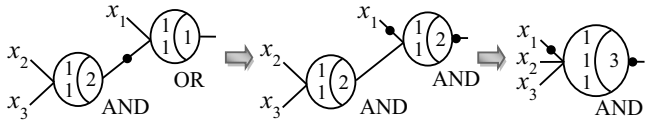

Fig. 5. An example of merging two threshold gates based on Transformation 2 with the De Morgan's laws.

In a TLN, there may exist some gates whose output is a constant value 0 or 1. These gates can be replaced with a constant value 0 or 1.

For a threshold gate, if the sum of all the weights is less than the threshold value, the output of the gate is a constant 0. On the other hand, if the threshold value is less than or equal to 0, the output of the gate is a constant 1. When a gate is replaced with a constant value, the value can be propagated forward to optimize other gates.

Transformation 2: Adjacent AND or OR gate merging

In a conventional Boolean logic network, two adjacent AND gates (or OR gates) can be merged to be a larger AND gate (or OR gate). Similarly, this transformation also works for a TLN.

Suppose g and g_f are two adjacent threshold gates and g_f is an fanin of g. If g and g_f are both AND gates (or OR gates), they can be merged to be a larger AND gate (or OR gate). Let g_r denote the resultant gate. The inputs of g_r are the union of the inputs of g except g_f and the inputs of g_f. All the weights in g_r are 1. If g_r is an AND gate, the threshold value is the number of inputs. If g_r is an OR gate, the threshold value is 1. When g_f only drives g, merging them decreases the overall gate count by 1.

Additionally, if there is a NOT gate in between g and g_f, and one of g and g_f is an AND gate and the other one is an OR gate, we can apply the De Morgan's laws[1] to either g or g_f to eliminate the NOT gate and make g and g_f the same type. Then, they can be merged.

Fig. 5 shows such an example. The De Morgan's laws are first applied to the OR gate to convert it into an AND gate and eliminate the NOT gate. Then, the two AND gates are merged to be a bigger AND gate.

The De Morgan's laws can be applied to the following transformations for creating more optimization opportunities as well.

Transformation 3: AND gate-based merging

Transformation 3 is adapted from the method in [9] for computing the weights and the threshold value of a Boolean function $h(x_1, x_2, ..., x_n)$, where $h(x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_{n-1})x_n$ and f is a threshold function.

For a threshold gate g, if g is an AND gate, it can be merged with one of its fanin gates g_f. The merging method is shown in Fig. 6(a) and (b). In Fig. 6(a), suppose that g has n inputs, $x_1 \sim x_{n-1}$ and g_f, and g_f has m inputs, $y_1 \sim y_m$. The weights of $y_1 \sim y_m$ are $w_{f\_1} \sim w_{f\_m}$ and the threshold value of g_f is T_f. The resultant gate g_r of merging g and g_f is shown in Fig. 6(b). g_r has $n-1+m$ inputs, $x_1 \sim x_{n-1}$ and $y_1 \sim y_m$. The weights of $y_1 \sim y_m$ remain the same. $x_1 \sim x_{n-1}$ all have a same weight w_r with a value of $\sum_{i=1}^{m} w_{f\_i} - T_f + 1$. The threshold value T_r is $(n-1) * w_r + T_f$.

[1] $\neg(P \wedge Q) \Longleftrightarrow (\neg P) \vee (\neg Q)$ and $\neg(P \vee Q) \Longleftrightarrow (\neg P) \wedge (\neg Q)$

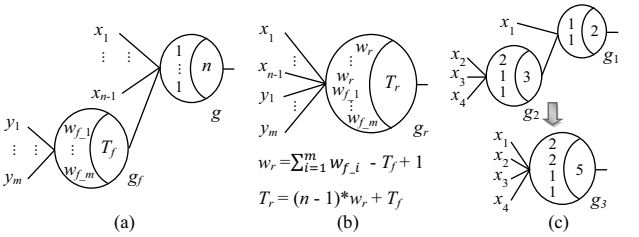

Fig. 6. Transformation 3. (a) Gates to be merged. g is an AND gate. (b) The resultant gate. (c) An example.

For example, in Fig. 6(c), the upper two gates, g_1 and g_2, can be merged to be the lower gate g_3. In g_3, the weights of $x_2 \sim x_4$ are the same with that in g_2. The weight of x_1 is 2, which equals $(2+1+1)-3+1$. Additionally, the threshold value of g_3 is 5, which equals $(2-1)*2+3$.

Let us explain how we derive this transformation. Consider Fig. 6(a) and (b) again. Because g is an AND gate, the weights of $x_1 \sim x_{n-1}$ are the same value. Thus, we assume that they have a same weight w_r in g_r as well. Additionally, the weights of $y_1 \sim y_m$ remain the same. Then, the problem of computing g_r is simplified to that of determining the values of w_r and T_r.

The output values of g and g_r must be equivalent for any input pattern. When $x_1 \sim x_{n-1}$ and the output value of g_f are 1, the output value of g is 1. For such an input pattern, in g_r, the possible minimum sum of the weights of the inputs with a value 1 is $(n-1)*w_r+T_f$. Thus, T_r must satisfy the Inequality (1) for g_r to output 1.

$$T_r \leq (n-1) * w_r + T_f \tag{1}$$

Furthermore, because g is an AND gate, when one of its input has a value 0, the output value of g is 0. Without loss of generality, we first suppose that the input with a value 0 is one out of $x_1 \sim x_{n-1}$. For such an input pattern, in g_r, the possible maximum sum of the weights of the inputs with a value 1 is $(n-2) * w_r + \sum_{i=1}^{m} w_{f\_i}$. Thus, T_r must satisfy the Inequality (2) for g_r to output 0. Next, we suppose that the input with a value 0 is g_f. For such an input pattern, in g_r, the possible maximum sum of the weights of the inputs with a value 1 is $(n-1)*w_r+T_f-1$. Thus, T_r must satisfy the Inequality (3) for g_r to output 0.

$$T_r > (n-2) * w_r + \sum_{i=1}^{m} w_{f\_i} \tag{2}$$

$$T_r > (n-1) * w_r + T_f - 1 \tag{3}$$

As a result, the valid w_r and T_r must satisfy Inequalities (1) $\sim$ (3) simultaneously. According to Inequalities (1) and (3), we obtain that T_r equals $(n-1)*w_r+T_f$. Then, by substituting T_r in Inequality (2) with $(n-1)*w_r+T_f$, we get another inequality $w_r > \sum_{i=1}^{m} w_{f\_i} - T_f$. Thus, we can set w_r to $\sum_{i=1}^{m} w_{f\_i} - T_f + 1$ for satisfying Inequality (2). Finally, we get $T_r = (n-1)*w_r+T_f$ and $w_r = \sum_{i=1}^{m} w_{f\_i} - T_f + 1$.

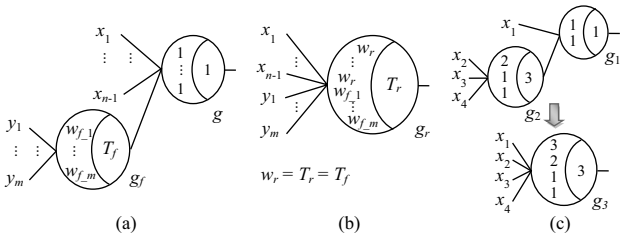

Fig. 7. Transformation 4. (a) Gates to be merged. g is an OR gate. (b) The resultant gate. (c) An example.

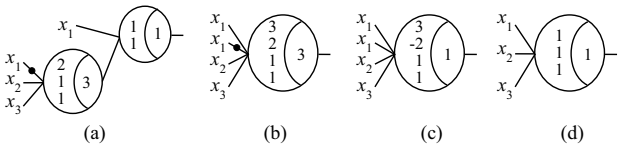

Fig. 8. An example of gate optimization by merging identical inputs.

Transformation 4: OR gate-based merging

Transformation 4 is derived according to the Theorem 2 in [18], which states that if $f(x_1, x_2, ..., x_l)$ is a threshold function, then $h(x_1, x_2, ..., x_{l+k}) = f(x_1, x_2, ..., x_l) + x_{l+1} + x_{l+2} + ... + x_{l+k}$ is also a threshold function, and presents a method for computing h.

Here, we formalize the theorem and the computation method as a transformation: For a threshold gate g, if g is an OR gate, it can be merged with one of its fanin gates g_f.

The merging method is shown in Fig. 7(a) and (b). Similarly, suppose that g has n inputs, $x_1 \sim x_{n-1}$ and g_f, and g_f has m inputs, $y_1 \sim y_m$. The resultant gate g_r of merging g and g_f has $n-1+m$ inputs, $x_1 \sim x_{n-1}$ and $y_1 \sim y_m$. The weights of $y_1 \sim y_m$ remain the same. $x_1 \sim x_{n-1}$ all have a same weight w_r, which is equivalent to the threshold value of g_f, T_f. The threshold value T_r is equivalent to T_f as well.

Fig. 7(c) shows an example of merging an OR gate g_1 and one of its fanin gates g_2 to be a bigger gate g_3. The weight of x_1 in g_3 and the threshold value of g_3 are equivalent to the threshold value of g_2.

Let us consider the correctness of this transformation. In Fig. 7(a) and (b), because g is an OR gate, when one out of $x_1 \sim x_{n-1}$ is 1, the output value of g is 1, and therefore the output value g_r must be 1. Thus, in g_r, the weights of $x_1 \sim x_{n-1}$ are all set to be equivalent to the threshold value. Furthermore, when all of $x_1 \sim x_{n-1}$ are 0, the output values of g and g_r must be identical as well. Thus, T_r is set to be equivalent to T_f and the weights of $y_1 \sim y_m$ remain the same.

In some cases, the resultant gate g_r needs to be further optimized. When g and g_f have common inputs, at least two inputs of g_r will be identical. If so, we merge the identical inputs to be one input. The merging method is as follows: If the polarities of the identical inputs are the same, we merge them to be one input and the weight of the input is the sum of the weights of the identical inputs. However, if their polarities are different, we need to make the polarities the same before we merge the inputs.

For example, the two gates in Fig. 8(a) have a common input x_1. After merging them, we get the resultant gate having two identical inputs with different polarities as shown in Fig. 8(b). Then, we make their polarities the same by applying the positive weight transformation in reverse to the input whose weight has a smaller absolute value as shown in 8(c). Finally, we merge the two inputs and get the resultant gate as shown in 8(d).

Transformation 5: Sum-of-product form to product-of-sum form conversion

Transformation 5 can work together with Transformation 3 to achieve more gate count reduction. First, let us use an example in Fig. 9 to demonstrate the motivation of Transformation 5.

In Fig. 9(a), the three gates g_1, g_2, and g_3 implement a Boolean function $x_1x_2 + x_2x_3$. Based on Transformation 4, g_3 can be merged with either g_2 or g_1. Then, the gate count decreases only by 1.

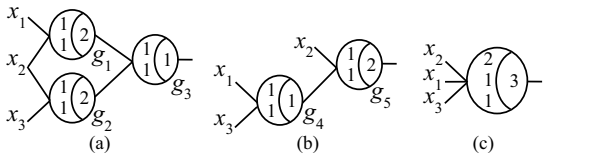

Fig. 9. An example of Transformation 5. (a) The original gates. (b) The resultant gates of applying Transformation 5. (c) The resultant gate of applying Transformation 3.

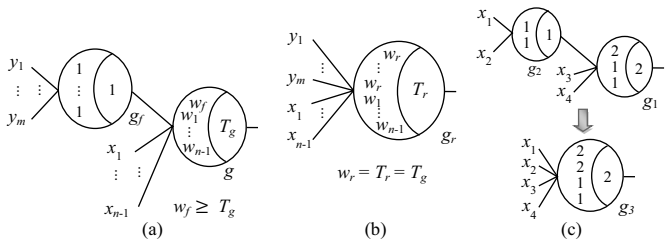

Fig. 10. Transformation 7. (a) Gates to be merged. g_f is an OR gate and a controlling-1 input of g. (b) The resultant gate. (c) An example.

However, g_1, g_2, and g_3 can be first restructured to become the two gates g_4 and g_5 in Fig. 9(b), which implement the equivalent Boolean function $x_2(x_1 + x_3)$. Then, g_4 and g_5 can be merged to be one gate based on Transformation 3 as shown in Fig. 9(c). Finally, the overall gate count decreases by 2.

Thus, for some gates, they can be pre-restructured for achieving more reduction. Transformation 5 is such a case of transforming a certain sum-of-product form into a product-of-sum form. It is also the distributive law in Boolean algebra.

When searching the opportunity for applying Transformation 5, we check if there exists an OR gate, at least two of whose fanins are AND gates and these AND gates have a common input. If so, they can be restructured.

Transformation 6: Product-of-sum form to sum-of-product form conversion

Transformation 6 is the duality of Transformation 5, which transforms a certain product-of-sum form into a sum-of-product form for achieving more reduction with Transformation 4.

For example, three gates implementing a Boolean function $(x_1 + x_2)(x_2 + x_3)$ can be transformed to be two gates, which implement the equivalent Boolean function $x_2 + x_1x_3$. Then, these two gates can be merged to be one gate based on Transformation 4.

To search the opportunity for applying Transformation 6, we check if there exists an AND gate, at least two of whose fanins are OR gates and these OR gates have a common input. If so, they can be restructured.

Transformation 7: Controlling-1 input-based merging

If a threshold gate has a controlling-1 input and the input is an OR gate, then they can be merged.

Let us use the example in Fig. 10 to demonstrate the transformation. In Fig. 10(a), the threshold gate g has n inputs, g_f and $x_1 \sim x_{n-1}$, and g_f is a controlling-1 input of g, where the weight of g_f is larger than or equal to the threshold value of g, T_g. Additionally, g_f is an OR gate and has m inputs, $y_1 \sim y_m$. Because g_f is an OR gate and it is an controlling-1 input of g, when one of the inputs of g_f is 1, g_f is 1 and g is 1 as well. Thus, g and g_f can be merged to be g_r as shown in Fig. 10(b). g_r has $m + n - 1$ inputs, $y_1 \sim y_m$ and $x_1 \sim x_{n-1}$. The weights of $y_1 \sim y_m$ are all equivalent to the threshold value of g_r, which is equivalent to T_g. The weights of $x_1 \sim x_{n-1}$ remain the same.

Fig. 10(c) shows an example of merging a gate g_1 and its controlling-1 input gate g_2, which is an OR gate, to be a bigger gate g_3. In g_3, the weights of x_1 and x_2 and the threshold value

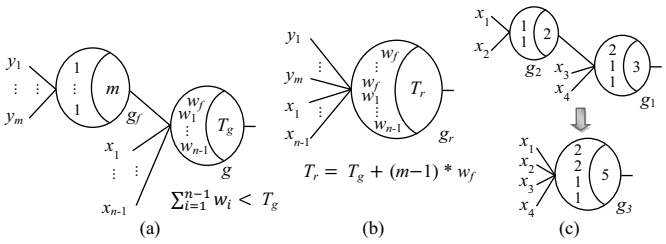

Fig. 11. Transformation 8. (a) Gates to be merged. g_f is an AND gate and a controlling-0 input of g. (b) The resultant gate. (c) An example.

of g_3 are all equivalent to the threshold value of g_1, which is 2. Additionally, the weights of x_3 and x_4 do not change.

Transformation 8: Controlling-0 input-based merging

Transformation 8 is similar to Transformation 7. If a threshold gate has a controlling-0 input and the input is an AND gate, then they can be merged.

Let us use the example in Fig. 11 to demonstrate the transformation. In Fig. 11(a), the threshold gate g has n inputs, g_f and $x_1 \sim x_{n-1}$, and g_f is a controlling-0 input of g, where the sum of the weights except that of g_f is less than the threshold value of g. Additionally, g_f is an AND gate and has m inputs, $y_1 \sim y_m$. Because g_f is an AND gate and it is a controlling-0 input of g, when one of the inputs of g_f is 0, g_f is 0 and g is 0 as well. Thus, g and g_f can be merged to be g_r as shown in Fig. 11(b). g_r has $m+n-1$ inputs, $y_1 \sim y_m$ and $x_1 \sim x_{n-1}$. The weights of $y_1 \sim y_m$ are all equivalent to the weight of g_f in g. Let w_f denote the weight value. The weights of $x_1 \sim x_{n-1}$ remain the same. The threshold value of g_r is equivalent to $T_g + (m-1) * w_f$, where T_g denotes the threshold value of g.

Fig. 11(c) shows an example of merging a gate g_1 and its controlling-0 input gate g_2, which is an AND gate, to be a bigger gate g_3. In g_3, the weights of x_1 and x_2 are all equivalent to the weight of g_2 in g_1, which is 2. Additionally, the weights of x_3 and x_4 are the same with that in g_1, which are 1. The threshold value of g_3 is equivalent to 5, the result of $3 + (2-1) * 2$, where 3 is the threshold value of g_1.

Similarly, in Transformations 7 and 8, if the two gates to be merged have common inputs, the resultant gate has at least two identical inputs, and thus, can be further optimized by merging the identical inputs.

B. Threshold logic optimization method

The overall optimization process consists of three rounds, each of which targets on certain transformations. At each round, each gate in the network is selected as a target gate one at a time in the topological order, and we check if a certain transformation can be applied to it and perform the transformation if applicable. With this selection order, a chain of AND and OR gates can be merged to be a threshold gate, such as the example in Fig. 2.

At the first round, we focus on Transformation 2 to merge adjacent AND or OR gates. Each target gate is checked if it can be merged with its fanout gates. If so, and the target gate drives only one gate, we perform the transformation. However, if the target gate has multiple fanout gates, we perform the transformations only when it can be merged with all of its fanout gates for making sure that the gate count can decrease.

Then, at the second round, we consider Transformations 5 and 6 to conduct the conversion between the sum-of-product form and the product-of-sum form. We perform the transformation only when it does not increase the gate count for making sure that the gate count can decrease at the third round. After the first two rounds, the TLN is still composed of AND or OR gates.

Finally, at the third round, we consider Transformations 3, 4, 7 and 8. First, for Transformations 3 and 4, we check whether a target gate is an AND or OR gate and has a fanin gate that only drives the target gate. If so, we merge them and consider the next target gate. Otherwise, we then consider Transformations 7 and 8. We check whether the target gate is an AND or OR gate and is a controlling input gate of the other gates, and they satisfy the sufficient conditions for Transformations 7 or 8. If so, we perform the transformation when the target gate drives only one gate and consider the next target gate. Otherwise, if the target gate has multiple fanout gates, we check whether it can be merged with all of its fanout gates based on all the applicable transformations. If so, we perform the transformations to eliminate the target gate.

During the optimization process, when two gates having common inputs are merged, we optimize the resultant gate and check whether it is a constant gate based on Transformation 1.

Although the overall optimization flow consists of three rounds, it is very efficient, because the searching process is a simple structure-matching process. The experimental results to be presented in the next section show that the overall optimization process for a large benchmark with 88854 gates costs less than one second.

Let us use the example in Fig. 4 to illustrate the optimization method. Fig. 4(b) shows the initial TLN to be optimized. At the first round, we consider Transformation 2 to search adjacent AND or OR gates for merging. In this example, no gates can be merged. Next, at the second round, we consider Transformations 5 and 6. Here, although g_3, g_4, and g_6 together can be converted to be in the product-of-sum form based on Transformation 5 with the De Morgan's laws applied to g_6, we do not perform the transformation. This is because g_3 has two fanout gates, the transformation increases the gate count. Thus, after the first two rounds, the TLN remains unchanged.

At the third round, according to Transformation 4, g_1 and g_5 are first merged to be the gate g_{1_5} as shown in Fig. 4(c) with the De Morgan's laws applied to g_5. Then, g_2 is merged with its two fanout gates g_3 and g_4 also based on Transformation 3 with the De Morgan's laws applied to g_2. Here, g_2 and g_3 are merged to be the gate g_{2_3} in Fig. 4(c). The resultant gate of merging g_2 and g_4 is further merged with g_6 to form the gate $g_{2\_4\_6}$ in Fig. 4(c) with the De Morgan's laws applied to g_6.

Finally, after optimization, all the NOT gates are integrated into their fanout gates and we obtain the resultant TLN in Fig. 4(d).

V. Experimental Results

We implemented the proposed method in C language within an ABC [2] environment. The experiments were conducted on a Linux workstation that comprises two Intel Xeon E5-2420 1.90GHz CPUs and 32GB memory. The benchmarks are from the IWLS 2005 benchmark suite [20] and we only consider their combinational portions.

Each benchmark in the AIG format is initially optimized by using the *resyn2* script in the ABC package. Then, it is transformed into a TLN by one-to-one mapping. After that, we optimize the TLN with the proposed optimization method and record the optimization results.

We also apply the ILP-based synthesis method [18] to synthesize each benchmark with the default setting for comparison, because it can directly synthesize an AIG-based Boolean logic network as well and is available at [21]. However, due to the license issue, we replace the default solver, *cplex*, with the *lp_solve* solver [22] in the ILP-based method. Based on the default setting, in each TLN obtained by the ILP-based method, a threshold gate can have at most 6 fanins. Thus, in the proposed method, we also set the same constraint for fair comparison. If a transformation will result in a threshold gate having more than 6 fanins, we do not perform the transformation.

To verify the optimization results, we finally transform each optimized TLN into a conventional Boolean logic network, and use the equivalence checking tool, *cec* [8], in the ABC package to verify the equivalence of the optimized and original benchmarks.

The experimental results are shown in Table I. Column 1 lists the benchmarks. Column 2 lists the number of nodes in each

TABLE I
THE SYNTHESIS RESULTS OF THE ILP-BASED METHOD AND OUR METHOD.

benchmark	N	ILP-based method [18]				our method				
		N	$int.$	$lev.$	T(s)	N	$ratio$	$int.$	$lev.$	T(s)
pci_conf.	84	91	175	6	2.2	62	0.68	156	4	0.0
stepper.	157	124	284	8	3.1	83	0.67	256	7	0.0
ss_pcm	172	173	349	6	4.4	135	0.78	308	5	0.0
usb_phy	357	287	659	8	7.2	221	0.77	604	7	0.0
sasc	563	461	1074	8	12.5	333	0.72	906	7	0.0
simple_spi	775	597	1429	10	16.1	436	0.73	1231	8	0.0
pci_spoci.	878	559	1483	15	15.6	399	0.71	1335	12	0.0
i2c	941	659	1620	12	18.1	482	0.73	1474	11	0.0
systemcdes	2641	2018	4758	21	57.7	1377	0.68	4122	19	0.0
spi	3429	2421	6092	23	75.6	1614	0.67	5368	19	0.0
des_area	4410	2774	7371	24	94.4	2011	0.72	6568	20	0.0
tv80	7233	4996	12967	37	191.1	3559	0.71	11193	30	0.1
mem_ctrl	8815	6573	15825	26	267.6	4721	0.72	13892	23	0.1
systemcaes	10585	7677	19168	36	334.4	5333	0.69	16005	33	0.1
ac97_ctrl	10395	8326	19437	8	330.0	6194	0.74	16671	7	0.1
usb_funct	13320	9860	24329	22	468.6	6842	0.69	20825	19	0.1
pci_bridge32	17814	13595	32875	23	769.9	10497	0.77	28570	21	0.2
aes_core	20509	14163	35307	18	761.2	10057	0.71	31091	17	0.2
wb_conmax	41070	28518	70823	15	2148.3	21956	0.77	64046	13	0.3
ethernet	57205	47004	109523	26	4978.9	35243	0.75	92796	23	0.6
des_perf	71327	59886	133711	16	7210.0	42719	0.71	117711	14	0.8
vga_lcd	88854	74095	176108	18	10918.6	55402	0.75	144453	17	0.9
average							0.72			
total					28685.6					3.4
ratio					8363.2					1

benchmark in the AIG format. It is also the number of threshold gates in the initial TLN obtained by one-to-one mapping. Columns $3 \sim 6$ respectively show the results of the ILP-based synthesis method. They are the number of threshold gates, the number of interconnections, the level, and the CPU time measured in seconds. Here, the number of interconnections is the sum of the fanin counts of all the threshold gates. For the benchmarks, *pci_conf.* and *ss_pcm*, the TLNs obtained by the ILP-based method have more gates than the original AIG-based networks. This is because there exist some POs directly driven by PIs in the benchmarks. Such a PO is counted as one gate in a TLN, but is not in an AIG-based network.

Columns $7 \sim 11$ show the corresponding results of the proposed method. Among them, Column 8 shows the ratios of the numbers in Column 7 with respect to that in Column 3. The ratios show the synthesis quality of the proposed method compared to the ILP-based method. For each benchmark, the CPU time shown in the last column includes the required time for transforming the benchmark into a TLN with one-to-one mapping and the time for optimizing the TLN.

For example, the last benchmark, *vga_lcd*, has 88854 nodes. By using the ILP-based method, the resultant TLN has 74095 gates and 176106 interconnections, and the level is 18. The consumed CPU time is 10918.6 seconds. However, by using the proposed method, the resultant TLN has only 55402 gates, approximately 75% of 74095. Additionally, there are only 144453 interconnections and the level is 17. The consumed CPU time is only 0.9 seconds.

The experimental results show that the proposed method can save an average of 28% threshold gates, compared to the ILP-based method. Additionally, every TLN obtained by the proposed method has fewer interconnections and a smaller level. Most importantly, the performance of the proposed method is much better than the ILP-based method. The proposed method spent less than 1 second synthesizing every benchmark. Furthermore, for overall benchmarks, the proposed method is 8363 times faster. Thus, the proposed method is a more efficient and effective synthesis method for TLNs.

In this work, we do not experimentally compare the proposed method with the BDD-based synthesis methods [4] [5], because they cannot directly synthesize an AIG-based Boolean logic network. A pre-process is required by them to transform an AIG into a network, which is composed of nodes with complex Boolean functions. Additionally, according to the experimental results in [5], which show that the BDD-based method can only save an average of 17% threshold gates, compared to the ILP-based method, the proposed method is competitive to the BDD-based method and even could be more efficient and effective.

VI. CONCLUSION

In this paper, we propose a simple but efficient and effective technique for TLN synthesis and optimization. A Boolean network in the AIG format is first transformed into a TLN by one-to-one mapping. Then, the TLN is optimized based on eight transformations for reducing the threshold gate count. The experimental results show that the proposed method is much more efficient and effective than a widely used synthesis method, which works based on time-consuming ILP solving.

This work dramatically speeds up the process of synthesizing a conventional Boolean logic network into a TLN with a better quality. When a compact TLN can be easily obtained, more and more possible applications of threshold logic would be developed. Thus, the proposed method could be an important technique for the compact logic representation to be promising.

REFERENCES

[1] M. J. Avedillo and J. M. Quintana, "A threshold logic synthesis tool for RTD circuits," in *Proc. Euromicro Symp. on Digital System Design*, 2004, pp. 624-627.

[2] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification," http://www.eecs.berkeley.edu/∼alanmi/abc/.

[3] H. A. Fahmy and R. A. Kiehl, "Complete logic family using tunneling phase logic devices," in *Proc. Int. Conf. Microelectron.*, 1999, pp. 22-24.

[4] T. Gowda and S. Vrudhula, "Decomposition based approach for synthesis of multi-level threshold logic circuits," in *Proc. Asia South Pacific Design Automation Conf.*, 2008, pp. 125-130.

[5] T. Gowda *et al.*, "Identification of threshold functions and synthesis of threshold networks," *IEEE Trans. Computer-Aided Design*, vol. 30, pp. 665-677, May 2011.

[6] Z. Kohavi, "*Switching and finite automata theory*," New York, NY: McGraw-Hill, 1978.

[7] P. Y. Kuo *et al.*, "On rewiring and simplification for canonicity in threshold logic circuits," in *Proc. Int. Conf. on Computer-Aided Design*, 2011, pp. 396-403.

[8] A. Mishchenko *et al.*, "Improvements to combinational equivalence checking," in *Proc. Int. Conf. on Computer-Aided Design*, 2006, pp. 836-843.

[9] S. Muroga, "*Threshold logic and its applications,*" New York, NY: John Wiley, 1971.

[10] A. Neutzling *et al.*, "Synthesis of threshold logic gates to nanoelectronics," in *Proc. Symp. on Integrated Circuits and Systems Design.* 2013, pp. 1-6.

[11] T. Oya *et al.*, "A majority-logic device using an irreversible single-electron box," *IEEE Trans. Nanotechnol.*, vol. 2, pp. 15-22, Mar. 2003.

[12] A. K. Palaniswamy and S. Tragoudas, "An efficient heuristic to dentify threshold logic functions," in *ACM J. Emerging Technol. in Comput. Systems*, vol. 8, no. 3, pp. 19:1-19:17, Aug. 2012.

[13] E. M. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," *in Proc. IEEE Int. Conf. Computer Design*, pp. 328-333, 1992.

[14] J. Subirats *et al.*, "A new decomposition algorithm for threshold synthesis and generalization of Boolean functions," *IEEE Trans. Circuits Systems I: Fundam. Theory Applicat.*, vol. 55, pp. 3188-3196, Nov. 2008.

[15] R. O. Winder, "Single stage threshold logic," *Switching Circuit Theory and Logical Design*, 1961, pp. 321-332.

[16] R. O. Winder, "Threshold logic," Ph.D. dissertation, Princeton University, Princeton, NJ, 1962.

[17] R. Zhang *et al.*, "Majority and minority network synthesis with application to QCA-, SET-, and TPL-based nanotechnologies," *IEEE Trans. Computer-Aided Design*, vol. 26, pp. 1233-1245, July 2007.

[18] R. Zhang *et al.*, "Threshold network synthesis and optimization and its application to nanotechnologies," *IEEE Trans. Computer-Aided Design*, vol. 24, pp. 107-118, Jan. 2005.

[19] R. Zhang *et al.*, "A method of majority logic reduction for quantum cellular automata," *IEEE Trans. Nanotechnol.*, vol. 3, pp. 443-450, Dec. 2004.

[20] http://iwls.org/iwls2005/benchmarks.html

[21] https://nanohub.org/resources/3353

[22] http://sourceforge.net/projects/lpsolve/