On Rewiring and Simplification for Canonicity in Threshold Logic Circuits

Pin-Yi Kuo, Chun-Yao Wang, and Ching-Yi Huang
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C.
s9862504@m98.nthu.edu.tw, wcyao@cs.nthu.edu.tw, s9862516@m98.nthu.edu.tw

***Abstract*—Rewiring is a well developed and widely used technique in the synthesis and optimization of traditional Boolean logic designs. The threshold logic is a new alternative logic representation to Boolean logic which poses a compactness characteristic of representation. Nowadays, with the advances in nanomaterials, research on multi-level synthesis, verification, and testing for threshold networks is flourishing. This paper presents an algorithm for rewiring in a threshold network. It works by removing a target wire, and then corrects circuit's functionality by adding a corresponding rectification network. It also proposes a simplification procedure for representing a threshold logic gate canonically. The experimental results show that our approach has 7.1 times speedup compared to the-state-of-the-art multi-level synthesis algorithm, in synthesizing a threshold network with a new fanin number constraint.**

I. Introduction

In past decades, design automation research targeting the functionality of VLSI circuits−such as logic synthesis, logic optimization, functional verification, and testing−have used Boolean logic representation almost exclusively. In comparison to threshold logic, however, Boolean logic representation requires a greater depth and more extensive nodes. For instance, a Boolean function $a(b+c)+bc$ can be realized by a single threshold logic gate. Additionally, the functionality of a threshold logic gate can be easily analyzed thanks to its special mechanism of output evaluation−the output value of a logic gate is evaluated by the relationship between its input-weighted summation and threshold value. For example, if an input vector 1100 produces an output 1 in a positive-weight threshold logic gate, other vectors $11--$ also produce output 1, where $-$ means don't care. This property facilitates logic synthesis and verification of threshold networks.

The development of threshold logic can be cast back to the 1960s. In 1962, an approximation method was proposed to determine the input weights and the threshold value of a threshold logic gate [30]. Later, linear programming and tabulation methods were proposed to determine whether or not a function could be realized in threshold logic [24]. The characteristics of threshold logic were explored and summarized in [15][24][30]. Although related research on threshold logic was conducted in the early days, threshold logic had a little impact on integrated circuit designs due to a lack of efficient implementation. Moreover, no efficient multi-level synthesis algorithm was proposed for threshold logic at that time, either. These reasons restricted the developments and applications of threshold logic.

CMOS implementations of threshold gates, however, have been developed recently [3∼4][6][29]. CMOS implementations of threshold gates have been a motivation to the investigation of the threshold logic. Furthermore, with the advances in nanotechnology, such as Resonant Tunneling Diodes (RTD) [1∼2][4][16][27], Quantum Cellular Automata (QCA) [4][16][28], and Single-Electron Transistor (SET) [4][10][16][22] that efficiently implement threshold logic, threshold logic now attracts more attention than before. Specifically, a MOnostable-BIstable transition Logic Element (MOBILE) can realize a threshold gate by using RTDs and Heterostructure Field-Effect Transistors (HFETs) [4][9][26]. This device achieves a realization with a shallower depth, fewer nodes, shorter wiring, and lower power consumption compared to the CMOS implementations [4].

This work was supported in part by the National Science Council of Taiwan under Grants NSC 99-2628-E-007-096, NSC 99-2220-E-007-003, NSC 100-2628-E-007-031-MY3, and NSC 100-2628-E-007-008.

In parallel with device technology advances, design automation research on threshold logic has also flourished. Synthesis methodologies for multi-level threshold networks have been proposed [17][18][31]. In the realm of verification, algorithms demonstrating equivalence checking for threshold networks have also been proposed [19][32]. Testing, an important issue in traditional Boolean logic, has also been addressed in its threshold logic counterpart. For example, a comprehensive study regarding the fault model and the Automatic Test Pattern Generation (ATPG) algorithm was proposed [20].

In traditional Boolean logic, rewiring is a logic restructuring technique that has been well developed and widely applied in the synthesis and optimization of VLSI circuits. Existing approaches to logic restructuring can be classified into two categories: *Redundancy* Restructuring and *Error-Injection-based* Restructuring. The Redundancy Restructuring approach keeps a circuit's functionality intact at *every* operation. *Node merging* [12∼14], *Rewriting* [25], and *Redundancy Addition and Removal* (RAR) [7∼8][11] belong to this category. On the other hand, the Error-Injection-based Restructuring approach transforms a circuit by first injecting an error/errors and then correcting it/them. *IRredundancy Removal and Addition* (IRRA) [23] belongs to this category.

In this work, we propose an error-injection-based rewiring algorithm for threshold logic networks. The problem formulation is as follows: Given an irredundant *target wire* to be removed from an *objective gate* in a threshold network consisting of threshold gates, our objective is to rectify the changed functionality of the original threshold network due to the target wire removal by adding some threshold logic gates at other locations. These added threshold logic gates are named the *rectification network*. Note that the proposed rewiring procedures are directly operated on a threshold network itself, and it does not need any information from its corresponding Boolean logic representation.

Two threshold gates with different appearances, such as different weights or threshold values, may have the same functionality. Minimizing these weights or threshold values reduces the implementation cost of the threshold gate [1][26]. In this work, we also propose a procedure to simplify the representation of a threshold gate such that the simplified representation is *canonical* meaning that any two functionally equivalent threshold gates will have the same representation. In other words, if two threshold gates cannot be represented with the same appearance after applying this procedure, they are functionally nonequivalent. Note that this simplification procedure of canonicity is designed for a single threshold gate rather than a threshold network.

In general, the applications of rewiring are for the synthesis and optimization of the threshold network, which is similar to traditional Boolean network rewiring[1]. In addition to traditional optimization objectives, the number of maximal fanins allowed for the gates in the threshold network also has to be determined before synthesis. If a network is going to be restructured with a smaller fanin

[1]It should be noted that this paper does not propose any algorithms focusing on any specific optimization objectives, e.g., area, timing, or power. Rather, it presents a theoretic foundation of the formal error-injection-based rewiring technique for threshold networks.

978-1-4577-1400-9/11/$26.00 ©2011 IEEE

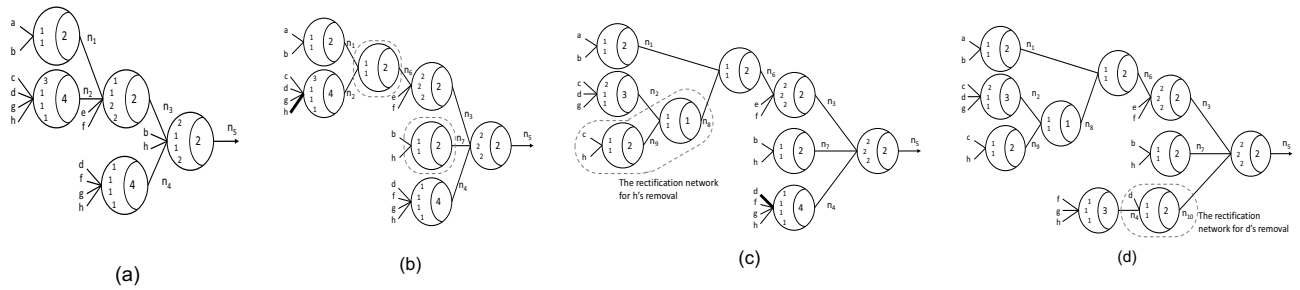


Figure 1: (a) The original threshold network. (b) The resultant network after input grouping and gate decomposition. (c) The resultant network after rewiring target wire h. (d) The resultant network after rewiring target wire d.

number[2], designers can either resynthesize the network by using the same synthesis methodology, or remove some wires from the gates violating this fanin number constraint and add the rectification network for correcting the functionality.

This work makes two main contributions:

1) It is the first rewiring algorithm employing on a threshold networks that changes its connectivity while preserving its functionality.
2) The proposed simplification procedure produces a canonical representation of a threshold gate.

The rest of the paper is organized as follows. Section II gives an example of our rewiring algorithm. Section III introduces the background. Section IV presents the proposed rewiring algorithm. Section V presents the simplification procedure. Section VI shows the experimental results of our rewiring algorithm. Finally, Section VII concludes this work.

II. An example for rewiring

In this section, we use a brief example to demonstrate the capability of our rewiring algorithm. We take a threshold network consisting of five threshold gates, as shown in Fig. 1(a). Here we assume the fanin number constraint of the network is four. If we want to produce another network with the same functionality but with a smaller fanin number constraint, e.g., three, we can rewire the network by using our algorithm instead of resynthesizing the whole network.

In the threshold network of Fig. 1(a), gates n_2, n_3, n_4, and n_5 violate this fanin number constraint. First, for gates n_3 and n_5, we can extract two new gates, n_6 and n_7, respectively, using the proposed gate decomposition method, as shown in Fig. 1(b). For gate n_2 in Fig. 1(b), we can remove target wire h. The rectification network n_8 is inserted at n_2's transitive fanout cone, as shown in Fig. 1(c). For gate n_4 in Fig. 1(c), we can remove target wire d. The rectification network n_{10} is inserted at n_4's fanout cone, as shown in Fig. 1(d).

Using our rewiring method, a threshold network with a new fanin number constraint is obtained. Previous threshold network synthesis tools resynthesized the network in order to satisfy different fanin number constraints [17][31]. Our rewiring algorithm, however, can achieve the same goal by focusing on single gates without resynthesizing the whole network.

III. Preliminaries

The section introduces definitions and some characteristics about threshold logic.

A. Threshold logic

A *linear threshold gate* (LTG) is an n binary inputs and one binary output function. The parameters of an LTG are weights $w_i; i = 1 \sim n$, which correspond to inputs $x_i; i = 1 \sim n$, and a threshold value T. A Boolean logic function is called a *threshold logic function* if and only if it can be realized as a single LTG. Furthermore, a threshold logic function may have many different threshold logic representations that are represented as *weight-threshold vectors* $\langle w_1, w_2, \ldots, w_n; T\rangle$. A network that is composed of LTGs is called a *threshold network*.

[2]This operation is trivial for traditional Boolean networks, but it is not the case for threshold networks.

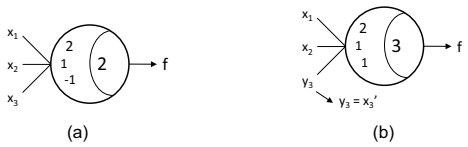


Figure 2: (a) An LTG implementing the function $f = x_1(x_2 + x_3')$. (b) The same threshold function $f = x_1(x_2 + x_3')$ after applying the positive-negative weight transformation.

The output f of an LTG is evaluated by EQ(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.

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

For example, in Fig. 2(a), the LTG with a weight-threshold vector $\langle 2, 1, -1; 2\rangle$ generates 1 if $2x_1 + x_2 - x_3 \geq 2$, and generates 0 otherwise.

The weights $w_i; i = 1 \sim n$ associated with corresponding inputs can be any real, positive, or negative numbers. However, these weights are usually integers due to technological considerations [9]. In this work, we assume the weights are integers for simplicity. In the last example, since $\{x_1 = 1, x_2 = 1\}$ or $\{x_1 = 1, x_3 = 0\}$ can make the LTG become 1, the Boolean function it represents is $f = x_1x_2 + x_1x_3' = x_1(x_2 + x_3')$. From this example, we can see that the threshold logic provides a more compact representation than traditional Boolean logic, with fewer nodes and a shallower depth.

Unateness is an important property of a threshold logic function, because all threshold logic functions are unate [21]. However, not all unate functions can be realized as threshold logic functions. If the weights of an LTG are all positive (negative), the function it represents is *positive* (*negative*) *unate*.

The rewired threshold network in this work is generated by an ILP-based approach [31] where each LTG is canonically represented. Given a unate function, an ILP formulation which describes its functionality as linear relationships searches the polytope vertices

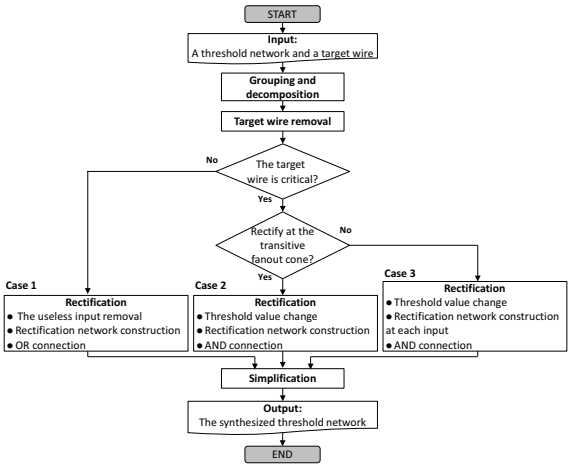


Figure 3: Our overall rewiring flow.

to find a point where the summation of the input weights and the threshold value in this LTG is minimal. Thus, this ILP-based approach will return a threshold network consisting of LTGs with minimal weights and threshold value. Furthermore, all input weights in a LTG range from 1 to T after applying the positive-negative weight transformation mentioned in the next subsection [15][30].

B. Positive-Negative weight transformation

Although the weights may be positive or negative integers in a general LTG, we make an LTG positive unate for easy analysis in this work. An LTG with some negative weights can always be transformed to a positive unate form by replacing these negative-weight input variables.

The transformation method for the negative weight in an LTG is described as follows [24]. First, one negative weight is negated into a positive one, and a new variable is set to the complement of the corresponding variable of this negative weight. Then, the threshold value is increased by the magnitude of the negative weight. These steps are repeated until all the weights are positive.

For example, let us transform an LTG $\langle 2, 1, -1; 2\rangle$ in Fig. 2(a) into a positive unate form. First, we negate the weight of x_3 from -1 to 1 and set a new variable $y_3 = x_3'$. Then, we add the magnitude of this negative weight 1 to the threshold value. As a result, the new representation of this threshold function becomes $\langle 2, 1, 1; 3\rangle$, as shown in Fig. 2(b).

IV. Rewiring for the threshold network

The section presents the proposed rewiring algorithm for threshold logic circuits. It consists of the input grouping and gate decomposition, the target wire removal, and the rectification network construction.

A. Overview

Fig. 3 gives an overview of our rewiring algorithm. The inputs are a threshold network and a target wire, the output is a functionally equivalent rewired threshold network. The *grouping* and *decomposition* are preprocessing stages operated on the threshold network. The proposed rewiring algorithm has different rectification methods with respect to the characteristics of target wires and rectification locations. At the end of our rewiring algorithm, the simplification procedure is performed to ensure that each threshold gate is canonically represented.

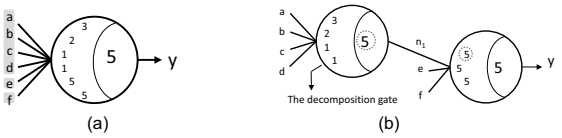


Figure 4: (a) Input grouping in an LTG. (b) The decomposition gate obtained from the input group $a \sim d$.

B. Input grouping and gate decomposition

Given an LTG, the grouping is a process that separates the inputs and its corresponding weights into different groups. The inputs of an LTG can be divided into one or more groups. With the aid of the grouping process, we can facilitate our rewiring process. Our grouping rule is as follows. First, we iteratively separate an input whose weight is equal to the threshold value of the objective gate as a single group. Second, the remaining inputs are separated as another group. We then treat the inputs of an LTG group-wise after this grouping process. That is, the inputs belonging to different groups are independently processed in our algorithm.

Next, we explain the reasoning behind this grouping rule. We know that the relationship between the weighted summation and the threshold value determines the output value of an LTG. If the weight of one input is equal to the threshold value, this input can independently change the output from 0 to 1. On the other hand, if the weight of an input is smaller than the threshold value, this input needs the weights from the other inputs to change the output from 0 to 1. Thus, we separate these inputs having smaller-than-the-threshold-value weights as a group. As a result, the inputs in different groups can be regarded as ORing together in this LTG.

By applying this grouping rule, we can easily decompose an LTG into more gates without changing this LTG's functionality. Specifically, each group can be extracted as a new LTG from the original LTG. The newly extracted LTG is named a *decomposition gate*. The extraction method is as follows. The threshold value of a decomposition gate is the same as that of the original LTG. The weight associated with a new decomposition gate in the original gate is also the threshold value because the new decomposition gate can determine the output value of the original gate without adding any weight from other inputs.

For example, given an LTG like the one, as shown in Fig. 4(a), inputs e and f are separated into two single groups after grouping because their weights are the same as the threshold value. The remaining inputs $a \sim d$ are separated into another group. Then, all these three groups can be individually extracted as new decomposition gates. In Fig. 4(b), we only extract the group consisting of inputs $a \sim d$. The threshold value of the decomposition gate and the weight associated with this new decomposition gate in the original gate, highlighted in dotted circles, are both 5.

C. Target wire removal

After the grouping and decomposition, a target wire is going to be removed. Here, we first introduce some terminology and properties related to the removal operation in this work. Then we discuss the potential results of the wire removal.

Definition 1: A single group LTG is *useless* if and only if it is an empty gate or it outputs zero for all input combinations.

Theorem 1: Given a nonempty LTG, it is useless if and only if it satisfies EQ(2), where n is the number of inputs in this gate.

$$\sum_{i=1}^{n} w_i < T \tag{2}$$

Due to the page limit, we omit all the proofs in this paper.

For example, in Fig. 4, assume that we would like to remove target wire a from the given LTG. After the grouping and decomposition, as shown in Fig. 4(b), we remove target wire a. Then the objective gate consisting of the inputs b, c, and d is useless according to Theorem 1. This is because the summation of $2, 1, 1$ in the decomposition gate $\langle 2, 1, 1; 5\rangle$ after removing a is less than 5.

Definition 2: An input in a single group LTG is *critical* if and only if this LTG will become useless after removing this input.

Theorem 2: Given a single group LTG, an input x_j with its corresponding weight w_j is critical if and only if it satisfies EQ(3), where n is the number of inputs in this gate.

$$\sum_{i=1,i\neq j}^{n} w_i < T \tag{3}$$

For example, in Figs. 4(a) and 4(b), input a is critical because the summation of weights $b \sim d$ is less than the threshold value. Similarly, inputs e and f are also critical because removing them results in empty decomposition gates, and an empty gate is useless by Definition 1.

Note that a critical input is important to the uselessness of a threshold gate. Furthermore, the functionality of a gate strongly depends on the relationship between the critical input and other inputs.

Definition 3: An input is *useless* if and only if the output of this LTG is intact when this input toggles under all input combinations.

Theorem 3: Given an input x_j with its corresponding weight w_j, x_j is useless if and only if it satisfies either EQ(4) or EQ(5) for each input combination, where n is the number of inputs in this gate.

$$\sum_{i=1,i\neq j}^{n} x_i w_i < T \text{ and } (\sum_{i=1,i\neq j}^{n} x_i w_i) + w_j < T \tag{4}$$

$$(\sum_{i=1,i\neq j}^{n} x_i w_i) + w_j \geq T \text{ and } \sum_{i=1,i\neq j}^{n} x_i w_i \geq T \tag{5}$$

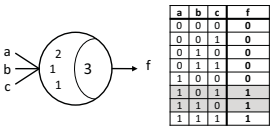
For example, in Fig. 4(b), input d will become useless after removing input c because it satisfies either EQ(4) or EQ(5) for all input combinations of a and b.

An objective gate will have two potential outcomes after we remove a target wire from it: a useless LTG or a normal LTG. If the objective gate becomes useless after removing the target wire, a dramatic functional loss occurs in the overall threshold network. To prevent the difficulty from this situation in our rectification scheme, we modify the threshold value of the objective gate if the objective gate becomes useless after this removal operation. The details about this modification will be addressed in the next section. Furthermore, since the removal operation may incidentally create useless inputs, which are not allowed in a normal LTG, we also remove them. On the other hand, if the objective gate is still a normal LTG after the removal operation, we do not change its threshold value.

D. Rectification network construction

In this section, we introduce the method of adding rectification networks at other locations to rectify the changed functionality of the original threshold network due to the target wire removal. Since the construction of the rectification network varies with the characteristics of the target wire, we analyze the relationship among the target wire and the other inputs, and divide the correction method into three cases with respect to the characteristics of a target wire, as seen in the flow of Fig. 3.

Definition 4: A single group LTG has a *critical-effect* if and only if there exists an assignment such that the output changes from 1 to 0 when each one of its inputs in this assignment changes from 1 to 0.



a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 5: An LTG and its critical-effect vectors.

Theorem 4: Given a single group LTG, the LTG has a critical-effect if it satisfies EQ(6), where n is the number of inputs in this gate.

$$\sum_{i=1}^{n} x_i w_i = T \tag{6}$$

An input assignment that satisfies the requirement of the critical-effect for an LTG is called a *critical-effect vector*. For example, in Fig. 5, input assignments 100 and 110 are the critical-effect vectors of LTG $\langle 2, 1, 1; 3\rangle$. This is because changing any 1 to 0 in these assignments will also change the output from 1 to 0. Note that EQ(6) is only a sufficient condition of Theorem 4. However, EQ(6) is also a necessary consition if the given LTG is obtained from an ILP-based synthesis algorithm [31]. That means all critical-effect vectors of the LTG satisfy EQ(6).

Case 1: The target wire is not critical: When the target wire is not critical, the remaining objective gate after the removal will not become useless. Thus, we preserve the functional relationship among the inputs in the remaining objective gate by keeping the threshold value intact. Since adding the rectification network at the transitive fanin cone of the objective gate will significantly affect the remaining functionality among other inputs, we only add the rectification network at the transitive fanout cone of the objective gate in this case.

The critical-effect vector mentioned above can be used to further analyze the functionality among all inputs of an LTG. Hence, we will use it in this case to construct the rectification network in our algorithm.

Let us first clarify the physical meaning of a critical-effect vector. In Fig. 5, the LTG has two critical-effect vectors 101 and 110. When considering 101, we find that another input assignment 111 also produces 1 after checking its truth table. This means that changing the second input b does not change the output value. Thus, if input b in this LTG is the target wire and has been removed, the remaining objective gate preserves the subfunction with respect to these two assignments 101 and 111. On the other hand, when considering another critical-effect vector 110, we find that vector 100 produces a different output, 0. Thus, if input b in this LTG is the target wire and has been removed, the remaining objective gate loses a subfuncion with respect to these two assignments 110 and 100.

In summary, we observe that the loss of a subfunction only occurs when removing a target input, which is assumed to be 1 in a critical-effect vector. Thus, to construct the rectification network, it is important and necessary to have information about the critical-effect vectors whose target input is assumed to be 1.

To search the critical-effect vectors of an LTG, we can exhaustively build its truth table and then find the input assignments satisfying EQ(6). However, this method is not scalable. Fortunately, thanks to the output evaluation mechanism in a positive-weight LTG, we can make this search process practical and efficient by deduction.

The method of rewiring for this case with the aid of the critical-effect vectors is described as follows. Given a single group LTG and a target wire x_t, we first remove any useless inputs after target wire removal. Second, we get all the critical-effect vectors of the LTG, where x_t is assumed to be 1. Third, we collect all the inputs that are assumed to be 1 in these critical-effect vectors. Then, the rectification

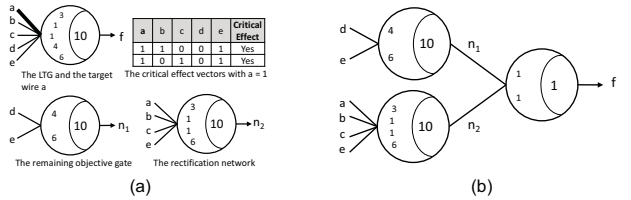


Figure 6: (a) The construction of rectification network for the removal of the target wire a. (b) The resultant threshold network after rewiring the target wire a.

network is constructed by using a new LTG consisting of the collected inputs in the third step with their corresponding weights and the threshold value of the original objective gate. Finally, we connect the remaining objective gate to this rectification network with an OR gate at its transitive fanout cone.

For example, in Fig. 6, given an LTG and the target wire a, inputs b and c become useless after the target wire a removal. Hence, we also remove them. Then, we get the critical-effect vectors which have a assumed to be 1, i.e., 11001 and 10101. Note that although 00011 is also a critical-effect vector, we do not count it in because $a = 0$. Then, we collect the inputs that are assumed to be 1 in these critical-effect vectors, 11001 and 10101; we get a, b, e from the first vector, and a, c, e from the second vector. Thus, the rectification network is constructed by using a new LTG consisting of these collected inputs, a, b, c, and e, with the weights and threshold value of the original objective gate, as shown in Fig. 6(a). Finally, the remaining objective gate is ORed with the rectification network, as shown in Fig. 6(b).

Next, let us explain why this method is correct for rectifying the functionality after a removal. As we mentioned, after the target wire removal, functional loss only occurs at the critical-effect vectors whose target input is assumed to be 1. Therefore, we first collect these vectors. These vectors then should be added back to the remaining objective gate to rectify the functionality. Thus, the rectification network is constructed in a way that has the same output as the original objective gate under these critical-effect vectors with an assumed target input of 1. As a result, these inputs assumed to be 1 in the critical-effect vector are collected because they caused the original objective gate to be evaluated as 1. If there are multiple critical-effect vectors, all inputs in these vectors assumed to be 1 are all collected. An LTG consisting of these inputs with the same weights and threshold value of the original objective gate is the rectification network. Then, this rectification network is added back by ORing it together with the remaining objective gate.

Case 2: The target wire is critical, and we rectify it at the transitive fanout cone: As mentioned in Definition 2, a critical input is an input whose removal results in a useless gate. Thus, if the target wire is critical, its value is always assumed to be 1 in all the critical-effect vectors. Hence, the whole function gets lost after the removal of a critical target wire, i.e., the remaining objective gate becomes useless. A useless gate is not allowed in our approach as mentioned in the previous section. Hence, to prevent the remaining objective gate from being useless, we will modify the threshold value during the rectification process, as detailed in the succeeding paragraphs.

The method of rewiring for this case is as follows. Given a single group LTG and a critical target wire x_t with the corresponding weight w_t, we decrease the threshold value in the remaining objective gate by w_t after removing x_t. Second, we construct the rectification network which is x_t only. Finally, we connect this rectification network to the remaining objective gate with an AND gate at its transitive fanout cone.

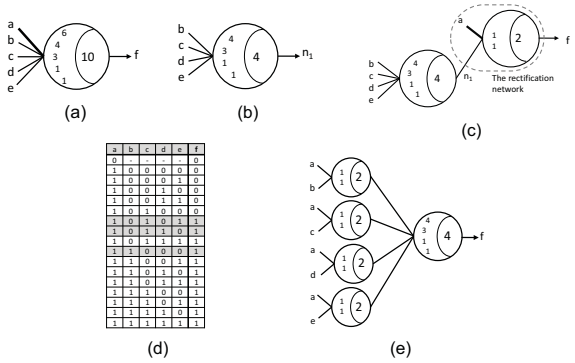


Figure 7: (a) The original objective gate and the target wire a. (b) The threshold value decrement after the target wire removal. (c) The resultant threshold network when the rectification network is added at the transitive fanout cone. (d) The truth table. (e) The resultant threshold network when the rectification network is added at the transitive fanin cone.

For example, given an LTG and the target wire a in Fig. 7(a), the threshold value is reduced to 4 from 10 after removal of a by decreasing the weight of a, as shown in Fig. 7(b). The rectification network is a only. Then, we connect this rectification network to the remaining objective gate using an AND gate, as shown in Fig. 7(c).

Next, let us explain why this rewiring procedure preserves the functionality of the original circuit. Since the original threshold network in this work is generated by the ILP-based synthesis method [31], a critical-effect vector always satisfies EQ(6) in Theorem 4. We have known that the target input is assumed to be 1 in all the critical-effect vectors. By the definition of critical-effect vector, an input assignment that has an x_t assumed to be 0 will produce 0, i.e., the input subspace with $x_t = 0$ always produce 0. However, after reducing the threshold value by w_t, the remaining objective gate could produce 1 when $x_t = 0$. Hence, we AND the x_t to the remaining objective gate such that when $x_t = 0$, the output is 0, which is consistent with the original functionality. In this rectification scheme, on the other hand, if the remaining objective gate produces 1, it implies that the weighted summation in this gate contributes the same amount of weight as it did in the original objective gate. Therefore, adding w_t when $x_t = 1$ will meet the requirement for producing 1 in the original objective gate.

For example, given an LTG $\langle 6, 4, 3, 1, 1; 10\rangle$ and the target wire a in Fig. 7(a), the LTG produces 0 under the input subspace of $a = 0$ by checking its truth table in Fig. 7(d). We find that all the critical-effect vectors 10101, 10110, and 11000 have a assumed to be 1. After reducing the threshold value from 10 to 4, the remaining objective gate produces 1 under its critical-effect vectors, 0101, 0110, and 1000. The rectification scheme is as shown in Fig. 7(c). In Fig. 7(c), if a is 0, the output is 0, which matches the subspace of $a = 0$ in the truth table of Fig. 7(d). For the assignments that produce 1 in n_1, they can add the weight of a to produce 1 in the output f when $a = 1$, which also match the truth table of Fig. 7(d).

Case 3: The target wire is critical, and we rectify it at the transitive fanin cone: In contrast to Case 2, we consider the rectification location at the transitive fanin cones of the target wire in this case. Since the target wire is still critical, we modify the threshold value to avoid a useless gate as we did in Case 2. The details have been addressed in Case 2.

The method of rewiring for this case is as follows. Given a single group LTG and a critical target wire x_t with the corresponding weight w_t, we decrease the threshold value of the remaining objective gate by

w_t after the removal. Second, we construct the rectification network that is x_t only. Finally, we connect this rectification network to each input, respectively, in the remaining objective gate with an AND gate at the transitive fanin cones.

For example, given an LTG and the target wire a in Fig. 7(a), the threshold value is reduced to 4 from 10 by decreasing the weight of a after the removal, as shown in Fig. 7(b). The rectification network is a only. Then, we connect this rectification network to each input in Fig. 7(b) using an AND gate, at the transitive fanin cones, as shown in Fig. 7(e).

The validity of this rectification can be explained in a similar manner as in Case 2. When x_t is a critical input, the output of the objective gate under an input assignment that has the x_t assumed to be 0 produces 0. Thus, after we AND x_t to each input in the remaining objective gate, the resultant network produces 0 under the vectors in the subspace of $x_t = 0$. When $x_t = 1$, the resultant network will act as the original gate with the setting $x_t = 1$.

For the same example in Figs. 7(a) and 7(e), when $a = 0$, the functionality in Fig. 7(a) is the same as that in Fig. 7(e). When $a = 1$, Figs. 7(a) and 7(e) also get the same result.

V. Simplification

After target wire removal and rectification network construction, the appearances of some LTGs in the threshold network may be changed such that they cannot be canonically represented. Thus, in this section, we introduce a simplification procedure that transforms a single group LTG to its canonical representation.

For example, given two LTGs $\langle 2, 1; 3\rangle$ and $\langle 1, 1; 2\rangle$, we recognize that both LTGs represent the same function $f(a, b) = ab$, because they both output 1 only at $\{a = 1, b = 1\}$. Since minimal weights and threshold value reduce the implementation cost of an LTG, it is desirable to minimize their values in an LTG [26]. An LTG is *canonical* if and only if it represents a function using minimal weights and threshold value. An LTG generated from the ILP-based synthesis method [31] is also canonical.

Next, we describe the simplification procedure as follows. First, a larger-than-1 common divisor divides the weights and the threshold value to get a more minimized representation if it exists. For example in Fig. 8(a), given an LTG $\langle 4, 4, 6, 8; 18\rangle$, a common divisor 2 divides the weights and the threshold value, as shown in Fig. 8(b).

Then, the weights and the threshold value of an LTG are gradually decreased while keeping the functionality intact. A decrement changes the functionality of an LTG is not allowed. To check if the functionality changes or not after a decrement in the weight and the threshold value, we only examine significant vectors that can exactly express the complete functionality of this LTG, rather than examine the whole truth table of this LTG. We will further discuss this idea later in the paper.

Next, we explain the method of decreasing weights and threshold value. If we decrease a unique weight by 1 in an LTG, the threshold value is decreased by 1 as well. However, we must simultaneously decrease the weights of inputs that have the same weight by 1 owing to their symmetrical property. That is, if the weights of these symmetric inputs become different after the decrement, the new representation is nonequivalent to the original one. Additionally, the corresponding threshold value is decreased by the number of 1 in these same-weight inputs of any critical-effect vector.

For example, in Fig. 8(b), the critical-effect vectors of the LTG $\langle 2, 2, 3, 4; 9\rangle$ are 0111 and 1011. Inputs a and b have the same weight 2, and the number of 1 in inputs a and b of the critical-effect vectors is 1. Thus, the weights of a and b are both decreased from 2 to 1, and these threshold value is decreased from 9 to 8, as shown in Fig. 8(c). The weight-decreasing operation is sequentially conducted and checked until each weight reaches 1.

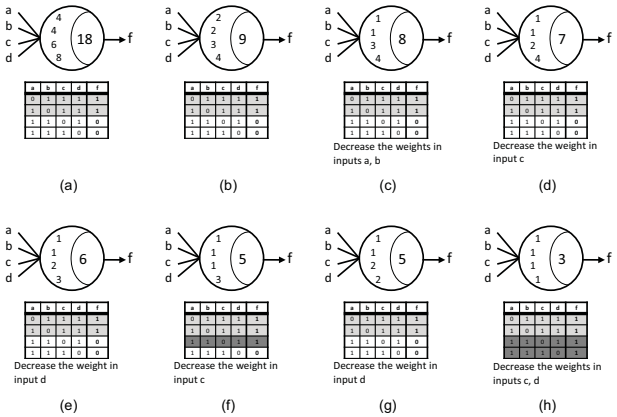


Figure 8: The simplification procedure for an LTG.

After each weight-decreasing operation, to verify if the functionality between the original LTG and the new LTG is intact or not is necessary. First, some terminology used in our functionality checking is introduced. A *subvector* of a vector is a vector whose input assumed to be 1 is the proper subset of this vector. A *supervector* of a vector is a vector whose input assumed to be 1 is the proper superset of this vector. A *brothervector* of a vector is a vector which has the same number of inputs assumed to be 1 as this vector.

For example, given a vector 011, its subvectors are 000, 001, and 010. Its supervector is 111. Its brothervectors are 101 and 110. To determine if the functionality of an LTG after a weight-decreasing operation is equivalent to the original LTG or not, we propose the method introduced in Theorem 5.

Theorem 5: Given two single group LTGs, they are functionally equivalent if and only if they produce the same outputs under all critical-effect vectors and the brothervectors of all critical-effect vectors.

Due to the special output evaluation mechanism of a positive-weight LTG, an LTG under the subvector of a critical-effect vector will output zero. Similarly, an LTG under the supervector of a critical-effect vector will output 1. Thus, if the critical-effect vectors for two LTGs are the same, their subvectors and supervectors will have the same outputs. However, the output of an LTG under the brothervectors of a critical-effect vector cannot be deduced by the output of the critical-effect vector. Thus, we also simulate the brothervectors of the critical-effect vectors. As a result, the outputs of the whole input space of an LTG can be derived by using the critical-effect vectors and their brothervectors.

In summary, after a weight-decreasing operation, if the outputs of new LTG under the critical-effect vectors and their brothervectors are the same as that of the original LTG, the functionality of the new LTG is intact by Theorem 5.

Now we use the same example to demonstrate this simplification procedure. For the LTG $\langle 2, 2, 3, 4; 9\rangle$, we first get its critical-effect vectors, 0111 and 1011, as highlighted, and their brothervectors. The output values under these assignments are shown in Fig. 8(b).

For each iteration of weight-decreasing operation, we decrease each input weight and the threshold value sequentially. Inputs a and b have the same weight; therefore, they are simultaneously decreased. The threshold value decrement for this situation has been addressed in the previous paragraph. Next, we check the validity of this decrement by comparing the outputs under these assignments, as shown in Fig. 8(c). Since these outputs are the same, this decrement is valid and a new representation $\langle 1, 1, 3, 4; 8\rangle$ is obtained. Next, the weight-decreasing operation for input c is shown in Fig. 8(d). After checking

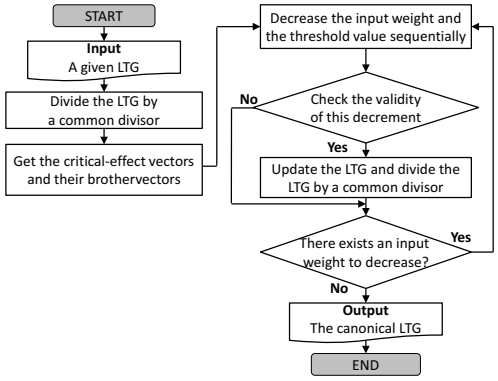


Figure 9: Our simplification procedure flow.

the validity, a new representation $\langle 1, 1, 2, 4; 7\rangle$ is obtained. Similarly, the decrement of input d is also valid and the LTG is updated as $\langle 1, 1, 2, 3; 6\rangle$, as shown in Fig. 8(e).

For the second iteration of weight-decreasing operation, since inputs a and b are the minimum positive integers, their weights cannot be decreased any more. Hence, the next weight-decreasing operation is for input c, as shown in Fig. 8(f). This decrement is invalid due to an output inconsistency under the assignment 1101, as highlighted. Finally, the weight-decreasing operation for input d is shown in Fig. 8(g). The updated representation $\langle 1, 1, 2, 2; 5\rangle$ is obtained.

In the last iteration, inputs c and d are simultaneously decreased by 1 and the threshold value is decreased by 2, since two 1s are assigned in inputs c and d of the critical-effect vectors, as shown in Fig. 8(h). Unfortunately, this decrement is invalid, either. Hence, we terminate the simplification procedure, and the canonical form of the original LTG $\langle 4, 4, 6, 8; 18\rangle$ is $\langle 1, 1, 2, 2; 5\rangle$.

The flow of the proposed simplification procedure is shown in Fig. 9. During our simplification procedure, checking the equivalence between the original LTG and updated LTG after a weight decrement is necessary. In traditional simulation-based equivalence checking, given an n-input LTG, we must use 2^n vectors to confirm whether or not the functionality is intact. However, thanks to the output evaluation mechanism of a positive-weight LTG, fewer vectors are sufficient. In the last example, only four vectors, or 25%, are needed for equivalence checking as compared to 16 vectors in traditional simulation-based verification[3].

VI. Experimental Results

We implemented our algorithm in C++ language. The experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.6). The benchmarks are from IWLS 2005 [33] in the blif format, and each benchmark is initially synthesized as a threshold network with a fanin number constraint, six, by using the method in [31]. The experiments consist of two parts: The first one shows the logic restructuring capability our rewiring algorithm offers. The second one shows the efficiency of our approach in resynthesizing a threshold network with a new fanin number constraint as compared to the-state-of-the-art multi-level synthesis algorithm [31].

In the first experiment, we rewired a threshold network using our rewiring algorithm. A target wire was randomly selected from an LTG in a threshold network whose input number is larger than two. After the removal, a rectification network was added with respect to the characteristics of the target wire. If the target wire is critical, we randomly select the rectification location at either the transitive fanin or fanout cones. Any added wire in a rectification network was not selected as a target wire candidate in the experiment. The simplification procedure was employed on changed LTGs such that the threshold network is canonically represented. To verify the correctness of our rewiring operation, the rewired threshold network was transformed to Boolean domain [19] and was compared against the original benchmark by using *verify* command in SIS [5].

The experimental results in the first experiment are summarized in Table I. Column 1 shows the names of the benchmarks. The next two columns show the total number of gates and wires in each benchmark. Column 4 shows the total times of rewiring operations on this benchmark. Column 5 shows the total CPU time on the rewiring. Column 6 shows the total CPU time on the simplification procedure measured in seconds. For example, in the *b20* benchmark, the total number of gates is 4431, and the total number of wires is 14020. It costs 8.57 seconds and 86.40 seconds to rewire and simplify this network 2562 times. According to Table I, the proposed rewiring and simplification procedure are very efficient. The simplification procedure costs more time because it needs to iteratively reduce the weights and threshold value for reaching a canonical form.

Table I
THE EXPERIMENTAL RESULTS OF REWIRING.

benchmark	|gate|	|wire|	|rewiring|	r_time$_{(s)}$	s_time$_{(s)}$
i2c	176	769	88	1.64	1.76
usb_phy	280	937	134	0.85	4.67
simple_spi	288	840	139	1.34	4.23
pci_spoci_ctrl	385	905	109	1.73	3.54
alu4	410	1407	210	2.66	6.57
s9234	554	1830	352	2.03	12.74
C3540	731	1688	387	2.08	13.56
dalu	810	2579	477	3.10	16.17
s13207	848	2235	450	2.76	15.32
C5315	879	2804	594	3.13	18.85
C6288	970	3485	394	2.40	13.10
rot	980	2878	610	3.35	19.43
C7552	1066	3886	701	4.48	25.62
tv80	1189	3485	614	3.80	20.35
spi	1646	4703	832	5.84	22.60
i10	1814	5893	1145	6.58	35.22
systemcdes	1907	4766	876	7.13	24.36
des	1920	5180	1008	6.37	31.49
aes_core	3417	13622	1552	8.60	47.02
mem_ctrl	3455	14655	2093	6.88	66.70
s38417	4280	20139	2915	12.07	93.43
b20	**4431**	**14020**	**2562**	**8.57**	**86.40**
ac97_ctrl	5732	17906	2560	9.48	82.55
b21	5844	13481	3495	10.20	110.40
usb_funct	6612	21613	2474	12.41	70.75
systemcaes	6885	22674	3656	15.43	102.80
s38584	6897	27750	2546	14.80	83.76
b22	7656	32771	3895	19.75	106.80
pci_bridge32	8344	29640	4983	21.14	134.30
b17	13460	39007	9140	47.82	215.45
wb_conmax	15719	47731	10872	70.05	232.70

In the second experiment, we demonstrate the efficiency of our rewiring algorithm for resynthesizing a threshold network with different fanin number constraints. The fanin number constraint for the original threshold network is six, and we want to reduce it to five. Instead of resynthesizing the whole network [31], our rewiring algorithm randomly removes an input in an LTG violating this new constraint, and then adds the corresponding rectification network. The simplification procedure is also applied after the rewiring. The equivalence between the two threshold networks after the rewiring is verified as well by using the same method in the first experiment.

Table II summarizes the results of the second experiment as compared to the-state-of-the-art [31]. Column 1 shows the names of the benchmarks. The next two columns show the total number of gates and wires, respectively, in this benchmark. Column 4 shows the total times of rewiring operations on this benchmark. Column 5 shows the total CPU time of resynthesizing this benchmark for satisfying a new fanin number constraint in [31]. Column 6 shows the total CPU time of our approach, and Column 7 shows the percentage

[3]The comparison ignores the efforts on searching the critical-effect vectors.

of CPU time reduction. For example, the *b20* benchmark has 4431 gates and 14020 wires. [31] requires 364.52 seconds to resynthesize the whole threshold network for meeting this new fanin number constraint, while our approach only costs 58.22 seconds to reach the same objective. The CPU time reduction is 84.0%.

According to Table II, our approach spent less CPU time, with a ratio of 63.3% reduction, compared to [31], in a benchmark on average. Furthermore, our approach is 7.1 times faster than [31]. This CPU time reduction increases with the growth of circuit size due to local, instead of global, resynthesis in our approach.

Table II
THE COMPARISON WITH THE-STATE-OF-THE-ART [31] FOR RESYNTHESIS.

benchmark	|gate|	|wire|	|rewiring|	[31] time$_{(s)}$	ours time$_{(s)}$	impr. %
i2c	176	769	28	8.60	3.24	62.3
usb_phy	280	937	55	2.54	1.67	34.3
simple_spi	288	840	37	2.48	1.84	25.8
pci_spoci_ctrl	385	905	43	2.17	2.08	4.1
alu4	410	1407	50	1.73	1.68	28.9
s9234	554	1830	74	5.62	2.85	49.3
C3540	731	1688	101	3.16	2.68	15.2
dalu	810	2579	52	5.06	1.73	65.8
s13207	848	2235	46	2.92	2.32	20.5
C5315	879	2804	192	21.50	9.46	56.0
C6288	970	3485	155	10.65	7.33	31.2
rot	980	2878	121	11.60	4.80	58.6
C7552	1066	3866	110	23.00	3.76	83.7
tv80	1189	3485	349	24.12	14.69	39.1
spi	1646	4703	663	67.85	25.22	62.8
i10	1814	5893	130	54.50	6.37	88.3
systemcdes	1907	4766	474	127.80	22.31	82.5
des	1920	5180	420	83.70	16.75	79.9
aes_core	3417	13622	792	402.60	38.73	90.4
mem_ctrl	3455	14655	1031	210.56	36.32	83.9
s38417	4280	20139	941	142.20	31.22	78.0
b20	**4431**	**14020**	**1463**	**364.52**	**58.22**	**84.0**
ac97_ctrl	5732	17906	1330	288.87	46.33	83.9
b21	5844	13481	1575	177.85	51.86	70.8
usb_funct	6612	21613	1405	293.26	42.13	85.6
systemcaes	6885	22674	1163	286.40	38.73	86.5
s38584	6897	27750	1981	525.72	83.74	84.1
b22	7656	32711	1595	320.04	52.04	83.7
pci_bridge32	8344	29640	1480	355.52	46.12	87.0
b17	13460	39007	2216	941.67	102.12	89.2
wb_conmax	15719	47731	2544	1386.34	108.24	92.2
total				6154.55	866.58	
average				198.53	27.95	**63.3**
ratio				**7.1**	**1**	

VII. CONCLUSION AND FUTURE WORK

This paper proposes a new rewiring technique for threshold networks. It works through the process of first removing a target wire and then correcting the functionality of the threshold network by adding its corresponding rectification network with respect to the characteristics of a target wire. It efficiently provides the capability of logic restructuring. A simplification procedure for canonicity that is directly applied to a single LTG is also proposed. When the threshold logic becomes a mainstream in the research of VLSI circuits, the contributions of this work will facilitate the applications of logic synthesis, verification, and various optimization goals.

REFERENCES

[1] M. J. Avedillo and J. M. Quintanaa, "A Threshold Logic Synthesis Tool for RTD Circuits," in *Proc. European Symp. on Digital System Design*, Sep. 2004, pp. 624-627.

[2] M. J. Avedillo, J. M. Quintana, H. Pettenghi, P. M. Kelly, and C. J. Thompson, "Multi-Threshold Threshold Logic Circuit Design Using Resonant Tunnelling Devices," *Electron. Lett.*, vol. 39, no. 21, Oct. 2003, pp 1502-1504.

[3] M. J. Avedillo, J. M. Quintana, A. Rueda, and E. JimCnez, "Low-Power CMOS Threshold-Logic Gate," *Electron. Lett.*, vol. 31, no. 35, Dec. 1995, pp. 2157-2159.

[4] V. Beiu, J. M. Quintana, and M. J. Avedillo, "VLSI Implementations of Threshold Logic-a Comprehensive Survey," in *Tutorial at Int. Joint Conf. Neural Networks*, 2003.

[5] Berkeley Logic Synthesis and Verification Group, "SIS: Synthesis of both synchronous and asynchronous sequential circuits," http://embedded.eecs.berkeley.edu/pubs/downloads/sis

[6] P. Celinski, J. F. Lopez, S. Al-Sarawi, and D. Abbott, "Low Power, High Speed, Charge Recycling CMOS Threshold Logic Gate," *Electron. Lett.*, vol. 37, Aug. 2001, pp. 1067-1069.

[7] S.-C. Chang, K.-T. Cheng, N.-S Woo, and M. Marek-Sadowska, "Postlayout Logic Restructuring Using Alternative Wires," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 587-596, June 1997.

[8] S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng, "Perturb and Simplify: Multi-level Boolean Network Optimizer," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1494-1504, Dec. 1996.

[9] K. J. Chen, K. Maezawa, and M. Yamamoto, "InP-Based High-Performance Monostable-Bistable Transition Logic Elements (MOBILE's) Using Integrated Multiple-Input Resonant-Tunneling Devices," *IEEE Eletron Device Letters*, vol. 17, pp.127-129, Mar. 1996.

[10] Y.-C. Chen, S. Eachempati, C.-Y. Wang, S. Datta, Y. Xie, and V. Narayanan, "Automated Mapping for Reconfigurable Single-Electron Transistor Arrays," *IEEE Design Automation Conf.*, 2011, pp. 878-883.

[11] Y.-C. Chen and C.-Y. Wang, "An Improved Approach for Alternative Wires Identification," in *Proc. Int. Conf. Computer Design*, 2005, pp. 711-716.

[12] Y.-C. Chen and C.-Y. Wang, "Fast Detection of Node Mergers Using Logic Implications," in *Proc. Int. Conf. on Computer-Aided Design*, 2009, pp. 785-788.

[13] Y.-C. Chen and C.-Y. Wang, "Node Addition and Removal in the Presence of Don't Cares," in *Proc. Design Automation Conf.*, 2010, pp. 505-510.

[14] Y.-C. Chen and C.-Y. Wang, "Fast Node Merging With Don't Cares Using Logic Implications," *IEEE Trans. Computer-Aided Design*, vol. 29, pp. 1827-1832, Nov. 2010.

[15] M. L. Dertouzos, *"Threshold Logic: A Synthesis Approach"*. Cambridge, MA: M.I.T. Press, 1965.

[16] D. Goldharber-Gordon, M. S. Montemerlo, J. C. Love, G. J. Opiteck, and J. C. Ellenbogen. "Overview of Nanoelectronic Devices," in *Proc IEEE*, vol. 85, no. 4, pp. 521-540, Jan. 1997.

[17] T. Gowda and S. Vrudhula, "Decomposition Based Approach for Synthesis of Multi-Level Threshold Logic Circuits," in *Proc. Asia and South Pacific Design Automation Conf.*, 2008, pp. 125-130.

[18] T. Gowda, S. Vrudhula, and G. Konjevod, "A Non-ILP Based Threshold Logic Synthesis Methodology," in *Proc. International Workshop on Logic and Synthesis*, 2007, pp. 222-229.

[19] T. Gowda, S. Vrudhula, and G. Konjevod, "Combinational Equivalence Checking for Threshold Logic Circuits," in *Proc. Great Lake Symp. VLSI*, March 2007, pp. 102-107.

[20] P. Gupta, R. Zhang, and N. K. Jha, "Automatic Test Generation for Combinational Threshold Logic Networks," *IEEE Trans. Computer-Aided Design*, vol. 16, pp.1035-1045, Aug. 2008.

[21] Z. Kohavi, *"Switching and Finite Automata Theory"*. New York, NY: McGraw-Hill, 1978.

[22] C. Lageweg, S. Cotofana, and S. Vassiliadis, "A Linear Threshold Gate Implementation in Single Electron Technology," in *Proc. IEEE Coput. Soc. Workshop VLSI*, 2001, pp. 93-98.

[23] C.-C. Lin and C.-Y. Wang, "Rewiring Using IRredundancy Removal and Addition," in *Proc. Design, Automation and Test in Europe*, 2009, pp. 324-327.

[24] S. Muroga, *"Threshold Logic and its Applications"*. New York, NY: John Wiley, 1971.

[25] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. Design Automation Conf.*, 2006, pp. 532-536.

[26] K. Maezawa, H. Matsuzaki, M. Yamamoto, and T. Otsuji, "High-Speed and Low-Power Operation of A Resonant Tunneling Logic Gate MOBILE," *IEEE Eletron Device Letters*, vol. 19, pp.80-82, March 1998.

[27] C. Pacha, P. Glosekotter, K. Goser, W. Prost, U. Auer, and F. Tegude, "Resonant Tunneling Device Logic Circuit," Dortmund/Gerhard-Mercator University of Duisburg, Germany, Tech. Rep., July 1999.

[28] M. Perkowski and A. Mishchenko, "Logic Synthesis for Regular Fabric Realized in Quantum Dot Cellular Automata," in *Proc. Int. J. Multiple-Valued Logic and Soft Comput.*, 2004, pp. 768-773.

[29] G. E. Sobelman and K.Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," in *Proc. Int. Conf. on Circuits and Systems*, vol. 2, 1998, pp. 61-64.

[30] R. O. Winder, *"Threshold Logic."* Ph.D. dissertation, Princeton University, Princeton, NJ, 1962.

[31] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, "Synthesis and Optimization of Threshold Logic Networks with Application to Nanotechnologies," in *Proc. Design Automation Test in Europe Conf.*, 2004, pp. 904-909.

[32] Y. Zheng, M. S. Hsiao, and C. Huang, "SAT-based Equivalence Checking of Threshold Logic Designs for Nanotechnologies," in *Proc. Great Lake Symp. VLSI*, May 2008, pp. 225-230.

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