Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies

Rui Zhang, *Student Member, IEEE,* Pallav Gupta, *Student Member, IEEE,* Lin Zhong, *Student Member, IEEE,* and Niraj K. Jha, *Fellow, IEEE*

***Abstract*— We propose an algorithm for efficient threshold network synthesis of arbitrary multi-output Boolean functions. Many nanotechnologies, such as resonant tunneling diodes (RTDs), quantum cellular automata (QCA), and single electron tunneling (SET), are capable of implementing threshold logic efficiently. The main purpose of this work is to bridge the current wide gap between research on nanoscale devices and research on synthesis methodologies for generating optimized networks utilizing these devices.**

While functionally-correct threshold gates and circuits based on nanotechnologies have been successfully demonstrated, there exists no methodology or design automation tool for general multi-level threshold network synthesis. We have built the first such tool, ThrEshold Logic Synthesizer (TELS), on top of an existing Boolean logic synthesis tool. Experiments with 56 multi-output benchmarks indicate that, compared to traditional logic synthesis, upto 80.0% and 70.6% reduction in gate count and interconnect count, respectively, is possible with the average being 22.7% and 12.6%, respectively. Furthermore, the synthesized networks are well-balanced structurally. The novelty of this work lies in the introduction of the first comprehensive synthesis methodology and tool for general multi-level threshold logic design.

***Index Terms*— Design automation, logic synthesis, QCA, RTD, threshold networks.**

I. Introduction

The Semiconductor Industries Association (SIA) roadmap [1] predicts that complementary metal-oxide semiconductor (CMOS) chips will continue to fuel the need for high-performance systems for another 10–15 years. However, advancements in electronic materials and devices have created nanoscale devices (RTDs, QCA, SETs, to name a few) that have novel structures and properties. Such devices offer the opportunity to further improve the compactness and speed of very large scale integrated (VLSI) systems. While it is easy to implement Boolean gates using CMOS, it is easier for many nanoscale devices to implement threshold gates.

As progress is made in the material and physical understanding of nanoscale devices, research must be done at the logic level to fully harness the potential offered by these devices. When CMOS was still in its infancy in the 1980s, researchers began to develop computer-aided design methodologies for it so that CMOS VLSI systems could be designed, synthesized, and tested in a reasonable time. The current CMOS dominance can partially be attributed to such developments. Today, nanotechnologies are in their infancy and the development of design automation methodologies for them is crucial if any of them is to be widely used. Among the various existing nanoscale devices [2]–[7], RTDs [4]–[6], QCA, and SET [7] are three promising nanotechnologies that are of particular interest to us because they implement threshold gates efficiently.

A threshold gate can be realized using RTDs and heterostructure field-effect transistors (HFETs), as shown by the circuit in Fig. 1(a). This circuit is called a monostable-bistable transition logic element (MOBILE) [8], [9]. A MOBILE is a rising edge-triggered, current-controlled gate. It consists of serially-connected load and driver RTDs. The RTD-HFETs connected in parallel to the load and driver RTDs perform a positive and negative weighting of the inputs, respectively. The output is logic 1 if the sum of the weighted inputs is greater than or equal to a threshold. Otherwise, it is logic 0.

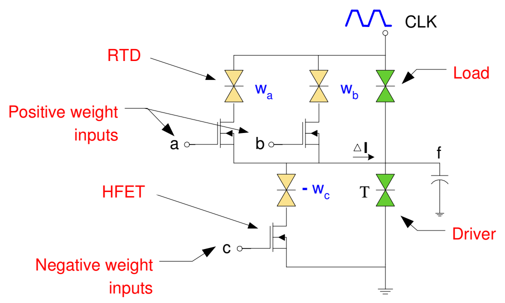
A QCA cell contains four quantum dots and two mobile electrons. Due to Coulombic interactions, the electron pair assumes one of the two configurations shown in Fig. 1(b). These configurations may be interpreted as digital states. A majority gate, also shown in Fig. 1(b), is a primitive gate in QCA that implements the function $M(A, B, C) = AB \vee BC \vee AC$. Majority gates are just a special case of threshold gates.

Circuits containing threshold gates have been demonstrated. The use of SET technology to implement such circuits has been described in [10], [11]. RTD-based threshold gates have been widely studied in [5], [8], [12]–[14]. However, the commercial application of threshold logic is very limited. The main reason is that the approach taken to design such circuits is a full-custom methodology. Furthermore, there exists no multi-level threshold network synthesis tool. As mentioned in [15], the usefulness of threshold gates will be determined not only by its availability, cost, and capabilities of its basic building blocks, but significantly more by the existence of automatic synthesis tools that could take advantage of them.

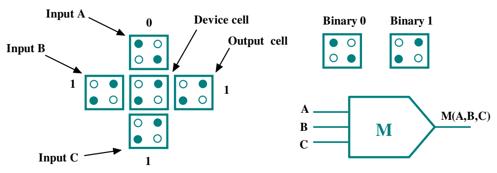
In this paper, we present the first comprehensive methodology for multi-level threshold logic synthesis and optimization from a Boolean logic description. Fig. 2 shows the CMOS and threshold logic design flows. Once a threshold network has been synthesized, it can be mapped onto a specific target nanotechnology. Our methodology takes into account defect tolerances in the input weights, and the fanin restriction on a threshold gate. Taking these parameters into account

Manuscript received February 28, 2004; revised August 25, 2004. This work was supported in part by NSF under grant No. CCR-0303789.

The authors are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA. (e-mail: {rzhang|pgupta|lzhong|jha}@ee.princeton.edu)



(a) A monostable-bistable logic element (MOBILE).



(b) A three-input QCA majority gate.

Fig. 1. Example of two nanotechnologies that implement threshold gates.

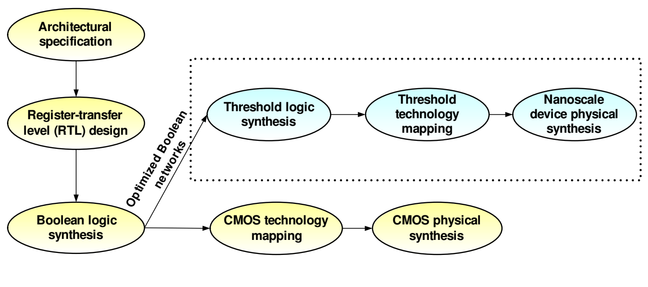

Architectural specification
Register-transfer level (RTL) design
Boolean logic synthesis
Optimized Boolean networks
Threshold logic synthesis
Threshold technology mapping
Nanoscale device physical synthesis
CMOS technology mapping
CMOS physical synthesis

Fig. 2. Conventional CMOS and threshold logic design flows.

improves the robustness of the synthesized network. While synthesizing, node sharing (*i.e.*, fanout node) is also preserved and thus, any advantage that is gained by preprocessing the network through a Boolean logic synthesis tool remains. The synthesized network is optimized in terms of gate count. The novel contributions of this work are as follows:

- This is the first *comprehensive* methodology for multi-level multi-output threshold network synthesis.
- Based on our methodology, we have built a threshold network synthesis tool on top of an existing Boolean logic synthesis tool.
- We formulate new theorems that describe properties of threshold logic and use them to our advantage in our methodology.

The remainder of this paper is organized as follows. In Section II, we present background material and discuss previous work in threshold logic synthesis. In Section III, we present an example to motivate the need for a threshold network synthesis methodology. In Section IV, we propose and prove several theorems on the properties of threshold logic. We then describe our synthesis methodology and its implementation in detail in Section V. We also discuss technology mapping based on MOBILEs in this section. In Section VI, we present our experimental results and conclude in Section VII.

II. Background and Previous Work

In this section, we describe some preliminary concepts to help the reader understand our proposed methodology better in later sections. Previous work in threshold logic synthesis is also presented in this section.

A. Threshold Logic

A *linear threshold gate* (LTG) has l two-valued inputs, $x_i \in \{0,1\}$, and a single two-valued output f. Its internal parameters are a threshold T and weights w_i, $i \in \{1,2,\dots l\}$, where weight w_i is associated with a particular input variable x_i. The input-output relation of an LTG is based on the fact that output f assumes the value 1 when the weighted sum of the inputs equals or exceeds the value of the threshold, T, and assumes the value 0 otherwise [16]. That is,

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

The weights and threshold of the LTG can be represented by the weight-threshold vector $\langle w_1, w_2, \dots, w_l; T\rangle$. If we want to increase the robustness of the LTG, we can incorporate defect tolerances into the definition as follows:

$$f(x_1,x_2,\dots,x_l) = \begin{cases} 1 & \text{if } \sum_{i=1}^{l} w_i x_i \geq T + \delta_{on} \\ 0 & \text{if } \sum_{i=1}^{l} w_i x_i \leq T - \delta_{off}, \end{cases} \quad (2)$$

where parameters δ_{on} and δ_{off} represent defect tolerances that must be considered since variations in the weights due to manufacturing defects and temperature changes can lead to malfunction. Generally, δ_{on} and δ_{off} take non-negative values. In the examples and synthesis results of this paper, we assume $\delta_{on} = 0$ and $\delta_{off} = 1$. However, our methodology and tool can take into account any user-specified values for δ_{on} and δ_{off}.

A Boolean logic function that can be realized by a single LTG is called a *threshold function* [16]. An LTG can be regarded as a generalization of conventional Boolean gates. An l-input NAND and NOR gate can both be realized by a single LTG. Because any Boolean logic function can be realized by a collection of only NAND gates or only NOR gates, such gates are called functionally complete. Hence, LTGs are also functionally complete. However, obviously not all functions can be realized by a single LTG. A network of threshold gates is called a *threshold network*. In the sequel, we will refer to LTGs as simply threshold gates.

B. Unateness

A Boolean logic function, $f(x_1,x_2,\dots,x_l)$, is said to be positive (negative) in variable x_i if there exists a disjunctive or conjunctive expression of f in which x_i appears in uncomplemented (complemented) form only. If f is either positive or negative in x_i, it is said to be *unate* in x_i. Otherwise, it is *binate* in x_i [17]. Unateness is an important property of threshold functions, because every threshold function is unate, but not vice versa [17].

C. Algebraically-Factored and Boolean-Factored Networks

A sum-of-products (SOP) expression, $f = \sum_{i=1}^{m} C_i$, is *algebraic* if no cube, C_i, is contained within another cube. That is, $\forall i, j,\ i \neq j,\ \ C_i \nsubseteq C_j$. An expression that is not algebraic is *Boolean* [18]. A factored form F is said to be algebraically-factored if the SOP expression obtained by multiplying F out directly, without using the identities $x\bar{x} = 0$ and $xx = x$, and single-cube containment (SCC), is algebraic [18]. Otherwise, F is Boolean-factored.

D. Linear Programming

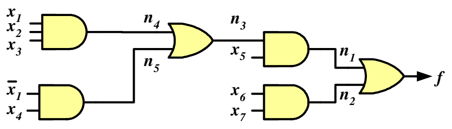
In linear programming, we have a $p \times q$ matrix $\mathbb{A}$, a $p \times 1$ vector $\mathbb{B}$, and a $1 \times q$ vector $\mathbb{C}$. We want to find a vector $\mathbb{X}$ of q elements such that the objective function $\mathbb{CX}$ is minimized subject to the p constraints given by $\mathbb{AX} \leq \mathbb{B}$. Integer linear programming (ILP) is a special case of linear programming that requires all of the elements in $\mathbb{X}$ to assume integer values.

E. Previous Work

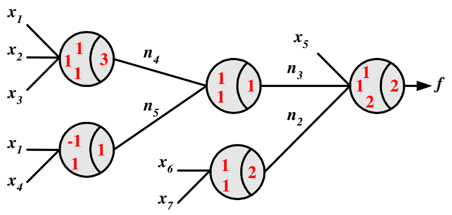
Research in threshold logic synthesis was done mostly in the 1950s and 1960s. However, at that time it was not easy to fabricate threshold gates and hence, the field failed to gain momentum in the computer engineering community. Nowadays, many competitive implementations of threshold gates are available, and a lot of experimental results from different applications have been published [8], [15]. Because of the need for a smoother transition towards logic design using nanotechnologies, we hope our work will reinvigorate this field.

In [19], [20], a series of relationships between the weights and the ON-set (set of cubes for which the function is 1) and OFF-set (set of cubes for which the function is 0) of a function was developed. Approximation methods were used to determine the weights of the inputs and the threshold of the function if the system of equations had a solution. In [17], unateness was presented as a necessary condition for a function to be threshold and admissible patterns on a Karnaugh map were used to determine whether a function is threshold or not. Unfortunately, because of their computational complexity, these methods are restricted to 10 or fewer variables. Linear programming and tabulation methods were used in [16] to determine if a function is threshold or not. However, multi-level threshold logic synthesis has not received much attention. Some existing methods perform synthesis by representing each product term in a sum-of-products (SOP) expression of a function as a threshold gate or by converting each gate in a Boolean network into a threshold gate (*i.e.*, one-to-one mapping). We will show that these methods lead to sub-optimal networks.

CMOS implementations of threshold gates can be found in [21]–[25]. A review of threshold logic can be found in [26] and a survey of VLSI implementations of threshold logic can be found in [15]. A multi-threshold logic circuit design using RTDs is described in [27]. In [28], a covering approach is used to synthesize two-level threshold networks. A satisfiability-based lattice synthesis algorithm is discussed in [29] for regular fabrics realized in QCAs.



(a) An example Boolean network.



(b) The equivalent synthesized threshold network.

Fig. 3. An example to motivate the need for a threshold logic synthesis methodology.

Another reason that multi-level threshold logic synthesis did not receive much attention before is that efficient algorithms to factorize a multi-level network were unknown at that time. Today, various algorithms exist to compute the kernels and co-kernels of a network which can then be used to perform algebraic or Boolean factorization [18], [30]. In addition, many methods have been developed for Boolean network simplification aided by the use of internal and external satisfiability don't care (SDC) and observability don't care (ODC) sets. Finally, tools, such as SIS [31], exist that can factorize and optimize a multi-level Boolean network.

III. Motivational Example

We present a motivational example to demonstrate the need for our threshold logic synthesis methodology in this section. The need will become apparent when we compare the gate count and the number of levels of the synthesized threshold network against its Boolean counterpart.

Consider the Boolean network shown in Fig. 3(a). This network contains seven CMOS gates and five levels (including the inverter). If we simply replace each gate with a threshold gate, the resulting threshold network will also contain seven threshold gates and five levels. However, this threshold network is sub-optimal because some nodes in Fig. 3(a) can be collapsed into a single threshold node. Choosing which node to collapse is critical. If we set the fanin restriction of a node to four, $f = n_1 \vee n_2$ can be collapsed to get $f = n_3x_5 \vee x_6x_7$.

Now, we must determine if f is a threshold function or not. One possible solution is to convert this problem into a linear programming formulation to determine if an optimal solution exists. In this case, it turns out that f is not a threshold function. Consequently, we must split f into two or more nodes. Efficient heuristics are required for splitting. We choose to split f as $f = n_3x_5 \vee n_2$ where $n_2 = x_6x_7$. Next, we proceed to synthesize n_3. After collapsing, n_3

can be expressed as $n_3 = x_1x_2x_3 \vee \bar{x}_1x_4$. This is not a threshold function. Therefore, we split n_3 into two nodes to get $n_3 = n_4 \vee n_5$, where $n_4 = x_1x_2x_3$ and $n_5 = \bar{x}_1x_4$. All of these nodes are threshold functions. The synthesized threshold network is shown in Fig. 3(b) and contains only five threshold gates and three levels. Each threshold gate in this network has the input weights shown on the left and the threshold shown on the right.

The above example demonstrates that a threshold logic synthesis methodology should try to address the following issues.

- It must be able to collapse the Boolean function of a node.
- It must determine if a Boolean function is threshold or not.
- If the function is not threshold, there should be an efficient way to split the function into smaller functions.
- Nodes that are shared in the original Boolean network should also be shared in the synthesized threshold network.

The way a non-threshold function is split is a key step in that it determines the quality of the synthesized network.

IV. Theorems For Threshold Logic

We present two theorems that describe properties of threshold logic in this section. We utilize them in our threshold logic network synthesis methodology. Along with the proof of each theorem, we demonstrate its application with an example.

Theorem 1: Given an expression for a unate Boolean function, $f(x_1, x_2, \ldots, x_l)$, replace literal x_i by literal $\bar{x}_j$, $i, j \in \{1, 2, \ldots, l\}$ and $i \neq j$, resulting in $g(x_1, x_2, \ldots, x_k)$, $k \in \{1, 2, \ldots, l\}$ and $k \neq i$. If g is not a threshold function, then f is not a threshold function.

Proof: We prove the contrapositive of the claim. That is, if f is a threshold function, then g is a threshold function. Assuming f is a threshold function with weight-threshold vector $\langle w_1, w_2, \ldots, w_l; T\rangle$, we have,

$$\sum_{k=1}^{l} w_kx_k \geq T + \delta_{on} \Rightarrow f = 1, \tag{3}$$

$$\sum_{k=1}^{l} w_kx_k \leq T - \delta_{off} \Rightarrow f = 0. \tag{4}$$

Note that Equations (3) and (4) represent 2^l inequalities for all value combinations of variables $x_1, x_2, \ldots, x_l$. By replacing x_i with $\bar{x}_j$, we obtain the following 2^{l-1} inequalities:

$$\sum_{k=1,k\neq i}^{l} w_kx_k + w_i\bar{x}_j \geq T + \delta_{on} \Rightarrow g = 1, \tag{5}$$

$$\sum_{k=1,k\neq i}^{l} w_kx_k + w_i\bar{x}_j \leq T - \delta_{off} \Rightarrow g = 0. \tag{6}$$

Since $\bar{x}_j = 1 - x_j$, we obtain,

$$\sum_{k=1,k\neq i,j}^{l} w_kx_k + (w_j - w_i)x_j \geq (T - w_i) + \delta_{on} \Rightarrow g = 1, \tag{7}$$

$$\sum_{k=1,k\neq i,j}^{l} w_kx_k + (w_j - w_i)x_j \leq (T - w_i) - \delta_{off} \Rightarrow g = 0. \tag{8}$$

If assignments for w_i and T exist such that the inequalities in Equations (3) and (4) are satisfied, then the inequalities in Equations (7) and (8) can also be satisfied with the weight-threshold vector, $\langle w_1, w_2, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{j-1}, w_j - w_i,$ $w_{j+1}, \ldots, w_l; T - w_i\rangle$. The variable sequence corresponding to the weights is $\langle x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{j-1}, x_j, x_{j+1},$ $\ldots, x_l\rangle$. Thus, g is also a threshold function. By proving the contrapositive, the proof is concluded. ■

As an application of Theorem 1, consider $f = x_1x_2 \vee x_3x_4$. To determine if f is threshold or not, we replace x_3 by $\bar{x}_1$. This results in $g = x_1x_2 \vee \bar{x}_1x_4$. Since g is binate in x_1, it is not a threshold function and, therefore, f is not a threshold function.

The relationship of the weights and threshold between functions containing x_i in positive and negative phase is given in [16]. That is, given a positive unate threshold function, $f(x_1, x_2, \ldots, x_l)$, with weight-threshold vector $\langle w_1, w_2, \ldots, w_l; T\rangle$, if x_i is replaced by $\bar{x}_i$ to get $g(x_1, x_2, \ldots, x_{i-1}, \bar{x}_i, x_{i+1}, \ldots, x_l)$, then the weight of x_i in g is simply $-w_i$. Furthermore, the threshold of g is $T - w_i$. We negate the original weight of each variable that appears in negative phase in g to get its new weight. To obtain the new threshold, we subtract the sum of the negated weights from the original threshold.

Theorem 2: If Boolean function $f(x_1, x_2, \ldots, x_l)$ is a threshold function, then $h(x_1, x_2, \ldots, x_{l+k}) = f(x_1, x_2, \ldots,$ $x_l) \vee x_{l+1} \vee x_{l+2} \vee \ldots \vee x_{l+k}$ is also a threshold function.

Proof: A threshold function can always be represented in positive unate form by substituting negative variables with positive variables. For simplicity, we assume that f is already expressed in this form. There exists a weight-threshold vector, $\langle w_1, w_2, \ldots, w_l; T\rangle$, for f since it is a threshold function. If any of the x_{l+j}, $j \in \{1, 2, \ldots, k\}$, equals 1, h equals 1. Otherwise, h is equal to f. If we set the weight w_{l+j} of x_{l+j} to a value no less than $T + \delta_{on}$, for example, $w_{l+j} = T + \delta_{on}$, then the output is 1 when x_{l+j} is 1. When x_{l+j} and some other x_i, $i \in \{1, 2, \ldots, l\}$, equal 1, the output is also 1. This is because we have represented the function in positive unate form, thereby guaranteeing that all the weights and threshold of f are positive [17]. When all of x_{l+j} equal 0, h equals f. Thus, a weight-threshold vector for h always exists. Therefore, h is a threshold function. ■

To illustrate Theorem 2, let $f_1(x_1, x_2, x_3) = x_1x_2 \vee x_1x_3$. Because f_1 is a threshold function with weight-threshold vector $\langle 2, 1, 1; 3\rangle$, $h_1(x_1, x_2, x_3, x_4, x_5) = x_1x_2 \vee x_1x_3 \vee x_4 \vee x_5$ is also a threshold function with weight-threshold vector $\langle 2, 1, 1, 3, 3; 3\rangle$. Taking another example, let $f_2(x_1, x_2) = x_1\bar{x}_2$. First, we represent f_2 in positive unate

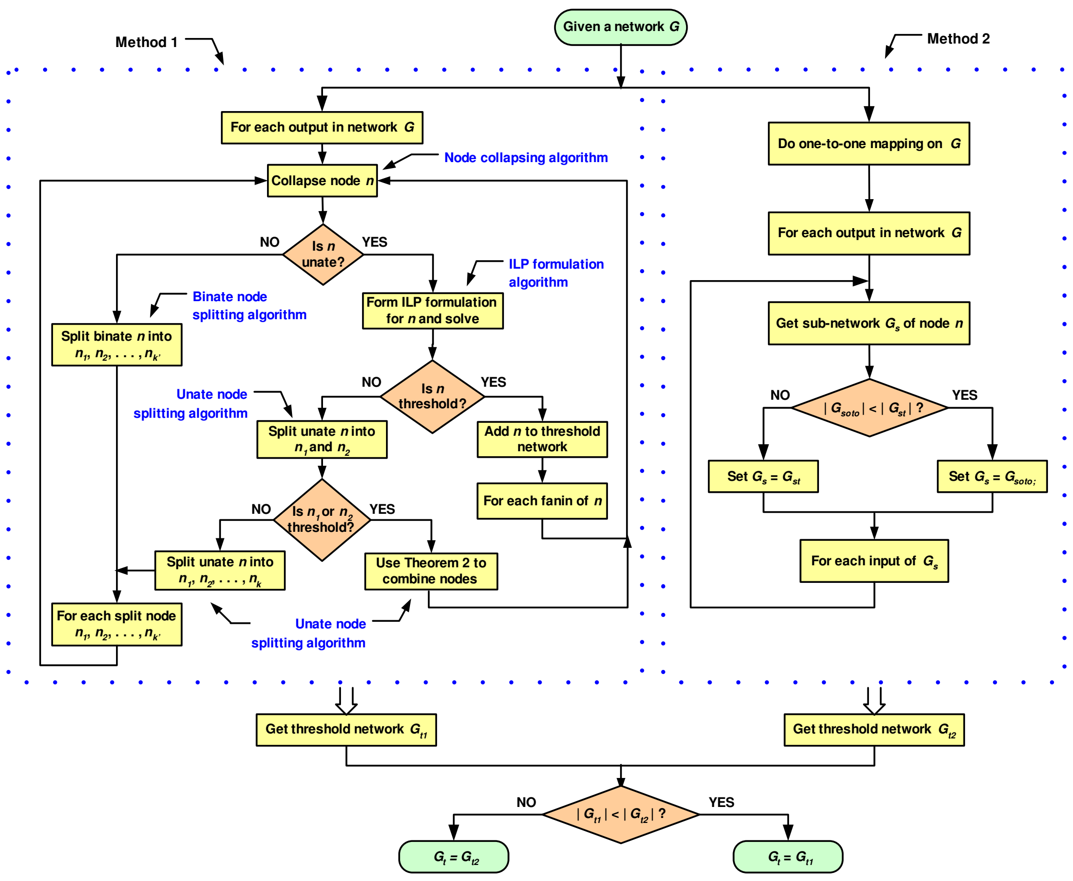

Given a network G
Method 1
Method 2
For each output in network G
Collapse node n
Node collapsing algorithm
Is n unate?
NO
YES
Form ILP formulation for n and solve
ILP formulation algorithm
Binate node splitting algorithm
Split binate n into n1, n2, . . . , nk'
Is n threshold?
Unate node splitting algorithm
Split unate n into n1 and n2
Add n to threshold network
For each fanin of n
Is n1 or n2 threshold?
Split unate n into n1, n2, . . . , nk
Use Theorem 2 to combine nodes
For each split node n1, n2, . . . , nk'
Do one-to-one mapping on G
For each output in network G
Get sub-network Gs of node n
| Gsoto | < | Gst | ?
Set Gs = Gst
Set Gs = Gsoto;
For each input of Gs
Get threshold network Gt1
Get threshold network Gt2
| Gt1 | < | Gt2 | ?
Gt = Gt2
Gt = Gt1

Fig. 4. Flow diagram providing an overview of our threshold logic synthesis methodology.

form as $g_2(x_1, y_2) = x_1 y_2$, where $y_2 = \bar{x}_2$. Since g_2 is a threshold function with weight-threshold vector $\langle 1, 1; 2\rangle$, $h_2(x_1, y_2, x_3) = g_2(x_1, y_2) \vee x_3 = x_1 y_2 \vee x_3$ is also a threshold function with weight-threshold vector $\langle 1, 1, 2; 2\rangle$. Since $y_2 = 1 - x_2$, $x_1\bar{x}_2 \vee x_3$ is also a threshold function with weight-threshold vector $\langle 1, -1, 2; 1\rangle$.

V. Methodology and Implementation

We discuss our multi-level threshold logic synthesis methodology and its implementation details in this section. The variables that are used in our flow diagram and algorithms are defined as follows:

G	A Boolean network.
$\|G\|$	The number of nodes in network G.
G_{soto}	The threshold sub-network obtained by one-to-one mapping.
G_{st}	The threshold sub-network obtained by our threshold synthesis Method 1.
P	Set of primary inputs in network G.
S	Set of fanout nodes in network G.
n	A node in network G.
F_n	Set of fanins of node n.
$x_{(j)}$	The (j^{th}) fanin of a node.
r	Fanin restriction on a threshold gate.
$\vec{v}$	Array of nodes.
K_n	Set of cubes of node n.
C_{n_i}	The i^{th} cube of node n.
$\overrightarrow{W}$	The weight-threshold vector, $\langle w_1, w_2, \ldots, w_l; T\rangle$.

A. Overview of the Method

Fig. 4 gives an overview of the main steps that comprise our methodology. The input to our methodology is an algebraically-factored multi-output combinational Boolean network, G, and its output is a functionally equivalent threshold network, G_t. An algebraically-factored Boolean circuit is used as an input because its nodes are more likely to be unate and hence possibly threshold functions. The user can specify the fanin restriction and defect tolerances for the threshold gates in the network.

Two methods are used for synthesis as shown in Fig. 4. In Method 1, the synthesis algorithm begins by processing each primary output of Boolean network G. First, the node representing a primary output is collapsed. If the node represents a binate function, it is split into multiple nodes which are then processed recursively. If the unate node is a threshold function, it is saved in the threshold network and the fanins of the node are processed recursively. Otherwise, the unate node is first split into two nodes. If either of the split nodes is a threshold function, Theorem 2 is used as a simplification step. If neither of the split nodes is a threshold function, the original node is split into multiple nodes which are then processed recursively. The synthesis algorithm by Method 1 terminates when all the nodes in network G are mapped into threshold nodes.

In Method 2, one-to-one mapping is first performed on network G. After that, starting from each primary output, we obtain the subnetworks. These subnetworks consist of

```
Require: node $n$, $r > 0$
    if $|F_n| > r$ then
        $\vec{v} \leftarrow$ split $n$ by one-to-one mapping method
3:      return $\vec{v}$ // array of split nodes
    $n' \leftarrow n$
    if not all fanins of node $n$ are primary inputs then
6:      while $|F_{n'}| \leq r$ do
            for every fanin $x$ of node $n'$ do
                // if neither primary input nor fanout node
9:              if $x \notin P \ \wedge \ x \notin S$ then
                    substitute the function of $x$ into $n'$
                    if $|F_{n'}| > r$ then
12:                     $n' \leftarrow$ undo the substitution of $x$ into $n'$
                        break
            // if all fanins are primary inputs or fanout nodes
15:         if $\forall x$ in $F_{n'}$, $x \in (P \cup S)$ then
                break
    return $n'$ // collapsed node
```

Fig. 5. The node collapsing algorithm.

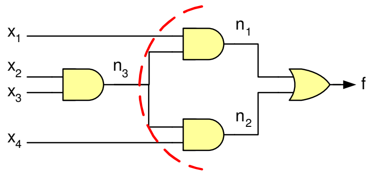


Fig. 6. Example network to demonstrate node collapsing on output f.

AND and OR gates only, and the number of primary inputs still satisfy the fanin restriction. Threshold synthesis is then performed on each subnetwork by Method 1. If the resulting threshold subnetwork has fewer threshold gates than the subnetwork obtained by the one-to-one mapping method, we choose this subnetwork. Otherwise, we choose the one-to-one mapped subnetwork. The synthesis algorithm by Method 2 terminates when all the subnetworks have been processed. Finally, we compare the two threshold networks obtained by these two methods, and pick the smaller one as the final threshold network. We describe each step in detail in the following subsections.

B. Node Collapsing

The node collapsing algorithm is shown in Fig. 5. Given a node n, if its number of fanins exceeds the fanin restriction, we split n by the one-to-one mapping method. Otherwise, we keep collapsing it until one of the following conditions is met:

- All fanins of n are primary inputs and/or fanout nodes;
- The fanin of n exceeds r.

To demonstrate node collapsing, consider the network with output node f in Fig. 6. Here, $f = n' = n_1 \vee n_2$, r is set to 4, $|F_{n'}| = 2$, $F_f = \{n_1, n_2\}$, $P = \{x_1, x_2, x_3, x_4\}$, and $S = \{n_3\}$. Since the inputs to f are not primary inputs and $|F_{n'}|$ is less than r, we first collapse on n_1 to get $f = n' = x_1 n_3 \vee n_2$. Now, $|F_{n'}| = 3$ and $F_f = \{x_1, n_2, n_3\}$. Since $|F_{n'}|$ is still less than r, we continue by collapsing on n_2 to get $f = n' = x_1 n_3 \vee n_3 x_4$. Now, we cannot collapse x_1 and x_4 since they are primary inputs. Furthermore, observing that n_3 is a fanout node, we do not collapse it either. Thus, the final result after collapsing is $f = x_1 n_3 \vee n_3 x_4$.

As demonstrated in the example, node sharing is preserved during node collapsing because the collapsing on one fanin stops once this fanin is a fanout node. This implicitly helps to maintain some of the original network structure and provides guidance for better network decomposition. The benefit is profound when the network contains many fanout nodes.

C. Formulating Synthesis as an ILP Problem

Once a node has been collapsed into a unate function, it is necessary to determine whether it is threshold or not. This can be done by formulating the problem as a linear programming (LP) problem and then solving it. However, in our implementation, we solve this problem by casting it in an ILP formulation. This yields integer weights and threshold which are easier to implement with RTD-HFETs (an LP formulation would, in general, yield non-integer weights and threshold). There are at most 2^l distinct cubes for a logic function of l variables and this leads to 2^l inequalities which represent the constraints (note that l cannot exceed the fanin restriction r, and hence is not a large number). However, many of these constraints are redundant. We have devised a simple method to eliminate redundant constraints which makes the ILP formulation smaller and possibly faster to solve (A similar method based on extremal vectors can be found in [16]). The algorithm for formulating the ILP problem and determining the weight-threshold vector for a threshold function is shown in Fig. 7. Even though in the worst case an ILP problem may take exponential time to solve, in practice it is efficiently solved because of small values of l for each threshold gate in the network.

The algorithm is best demonstrated by an example. Given a unate function, $f(x_1, x_2, \ldots, x_l)$, if it contains variables in negative phase, we first transform these variables into other variables in positive phase using variable substitution. Consider $f = x_1\bar{x}_2 \vee x_1\bar{x}_3$, where x_2 and x_3 are in negative phase. By replacing $\bar{x}_2$ with y_2 and $\bar{x}_3$ with y_3, we get the positive unate function $g = x_1 y_2 \vee x_1 y_3$. The ILP formulation for g is as follows:

$$\begin{aligned} minimize: & \ w_1 + w_2 + w_3 + T && (9)\\ subject\ to: & \ w_1 + w_2 \geq T + \delta_{on} && (10)\\ & \ w_1 + w_3 \geq T + \delta_{on} && (11)\\ & \ w_2 + w_3 \leq T - \delta_{off} && (12)\\ & \ w_1 \leq T - \delta_{off} && (13)\\ & \ w_i \geq 0, \ integer, \ i = 1, 2, 3. && (14) \end{aligned}$$

The objective function for this ILP problem is defined as the summation of the weights and threshold. Since g has been transformed into a positive unate form, only 1 and don't care $(-)$ will appear in its ON-set cubes. The ON-set cubes of g are $(1\ 1\ -)$ and $(1\ -\ 1)$, where x_1, y_2, and y_3 is the variable sequence. To transform the ON-set cubes into inequalities, the i^{th} 1 value corresponds to weight w_i. We need not consider don't cares in the inequalities, because they represent redundancies in g. For example, the ON-set cube $(1\ 1\ -)$ corresponds to two inequalities, namely, $w_1 + w_2 \geq T + \delta_{on}$ and $w_1 + w_2 + w_3 \geq T + \delta_{on}$. The second inequality is redundant because once the first inequality is satisfied, the second inequality is automatically satisfied as well. Similarly,

```
Require: positive unate node $n$
    $n_p \leftarrow n$
    $n_n \leftarrow$ invert $n$
 3: for $i = 1$ to $|F_n|$ do // objective function
        print "$w_i+$"
    print "$T$"
 6: for $i = 1$ to $|K_{n_p}|$ do // ON-set inequalities
        for $j = 1$ to $|F_p|$ do
            if $x_j \in C_{n_{p_i}}$ then
 9:             print "$w_j+$"
        print "$-\delta_{on} \geq T$"
    for $i = 1$ to $|K_{n_n}|$ do // OFF-set inequalities
12:     for $j = 1$ in $|F_n|$ do
            if $x_j \notin C_{n_{n_i}}$ then
                print "$w_j+$"
15:     print "$\delta_{off} \leq T$"
    $\overrightarrow{W} \leftarrow$ solve ILP problem
    if $\overrightarrow{W} \neq \emptyset$ then
18:     for $j = 1$ in $|F_n|$ do
            if $x_j$ in negative phase in $n$ then
                // subtract weight from original threshold
21:             $W[T] \leftarrow W[T] - W[w_j]$
                $W[w_j] \leftarrow -W[w_j]$ // negate the weight
        return $\overrightarrow{W}$ // weight-threshold vector
24: else
        return $\emptyset$ // no solution exists
```

Fig. 7. The ILP formulation algorithm.

$w_1 + w_3 \geq T + \delta_{on}$ represents the constraint imposed by the cube $(1\ -\ 1)$.

To compute the OFF-set cubes of g, we simply invert g to get $\bar{g}$. The ON-set cubes of $\bar{g}$ correspond to the OFF-set cubes of g. Because $\bar{g}$ is always in negative unate form, only 0 and $-$ appear in its ON-set cubes. Continuing with the earlier example, we invert g to get $\bar{g} = \bar{x}_1 \vee \bar{y}_2\bar{y}_3$ after simplification. The ON-set cubes for $\bar{g}$ are $(0\ -\ -)$ and $(-\ 0\ 0)$. For the OFF-set inequalities, the i^{th} don't care corresponds to weight w_i. Therefore, the ON-set inequalities for $\bar{g}$ are $w_2+w_3 \leq T-\delta_{off}$ and $w_1 \leq T - \delta_{off}$. Thus, the OFF-set inequalities for g are $w_2 + w_3 \leq T - \delta_{off}$ and $w_1 \leq T - \delta_{off}$, as given in Equations (12) and (13).

By requiring the variables to be integer-valued (*i.e.*, constraint (14)), this ILP problem has an optimal solution. The weight-threshold vector for g is $\langle 2, 1, 1; 3\rangle$. Using the relationship mentioned in Section IV, the final weight-threshold vector for f is $\langle 2, -1, -1; 1\rangle$.

D. Unate Node Splitting and Combining

If the ILP problem for node n does not have a solution, the node must be split into multiple nodes to increase the likelihood of the split nodes being threshold functions. Fig. 8 outlines the splitting process of a unate node. How a unate node is split is contingent upon one of the following conditions:

1) All of the variables appear exactly once.
2) Some of the variables appear in all the cubes.
3) The most frequent variable(s) does not appear in all the cubes.
4) A tie between the most frequent variables is broken randomly.

If all the variables appear only once, we simply split the node into two, with each node containing roughly equal number of

```
Require: unate node $n$, $r > 0$
    if all variables appear once then // condition 1
        $n = n_1 \vee n_2$ s.t. $|K_{n_1}| = |K_{n_2}|$
 3: else if $\forall c, c \in K_n$, variable $x_i \in c$ then // conditions 2 and 4
        $n_1 \leftarrow x_i$ // assuming correct phase
        $n_2 \leftarrow$ factor $n$ w.r.t. $n_1$
 6: else // condition 3
        $x_i \leftarrow$ most frequently appearing variable
        $n_1 \leftarrow \sum c, x_i \in c \wedge c \in K_n$
 9:     $n_2 \leftarrow \sum c_1, c_1 \notin n_1 \wedge c_1 \in K_n$
    if $n_1$ is a threshold function then // assuming $|K_{n_1}| \geq |K_{n_2}|$
        combine nodes according to Theorem 2
12:     $\vec{v} \leftarrow n_1 \vee n_2$
    else if $n_2$ is a threshold function then
        combine nodes according to Theorem 2
15:     $\vec{v} \leftarrow n_1 \vee n_2$
    else
        $k \leftarrow (|K_n| \leq r)\ ?\ |K_n| : r$ // pick the smaller one
18:     $\vec{v} \leftarrow \sum_{i=1}^{k} n_i$ // split $n$ into $k$ smaller nodes
    return $\vec{v}$ // array of split nodes
```

Fig. 8. The unate node splitting algorithm.

cubes. When a variable appears in all the cubes, we split the node into two by factoring this variable out of the node. If the above two conditions are not met, we split the node using the most frequently appearing variable. For example, given $n = x_1x_2 \vee x_1x_3 \vee x_4x_5$, we split on x_1 to get $n_1 = x_1x_2 \vee x_1x_3$ and $n_2 = x_4x_5$, with $n = n_1 \vee n_2$. This last condition reduces the likelihood of a function being non-threshold because there are fewer candidate variables to choose from in the split nodes to prevent the condition in Theorem 1 from being satisfied.

Once a node has been split, we choose the larger node (*i.e.*, the one with more cubes) and check that node to see if it is a threshold function. If it is, we apply Theorem 2. Looking at the last example, since $n_1 = x_1x_2 \vee x_1x_3$ is a threshold function with weight-threshold vector $\langle 2, 1, 1; 3\rangle$, function $n = x_1x_2 \vee x_1x_3 \vee n_2$ is also a threshold function with the weight-threshold vector $\langle 2, 1, 1, 3; 3\rangle$. Now, n_2 is processed by further collapsing, threshold checking, or splitting. If neither of the split nodes is a threshold function, the original node is split into k smaller nodes, where k is the smaller of r and $|K_n|$. After splitting, $n = n_1 \vee n_2 \vee \cdots \vee n_k$, which is a threshold function with weight-threshold vector $\langle 1, 1, \ldots, 1; 1\rangle$. The split nodes, n_i, $i \in \{1, 2, \ldots, k\}$, are then processed recursively.

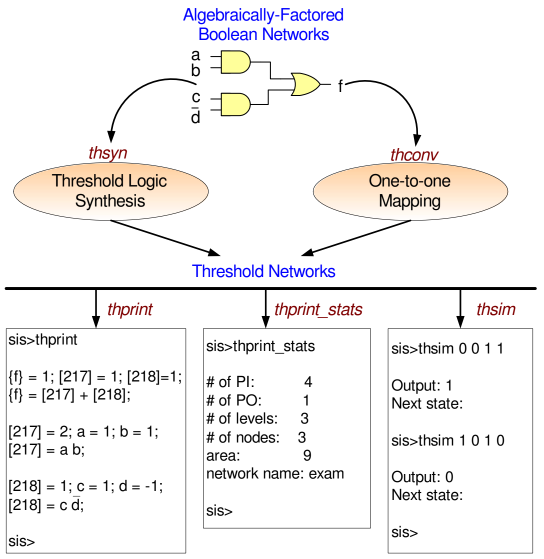
E. Binate Node Splitting

If a node n is binate, we split it into at most k nodes where k is the smaller of r and $|K_n|$. The splitting stops when a unate node is generated. The algorithm for binate node splitting is shown in Fig. 9. We first split the binate node on the most frequently appearing binate variable. If the split nodes are binate, we repeat the process. Otherwise, we stop splitting.

To demonstrate binate splitting, consider $n = \bar{x}_1x_4 \vee x_2x_3 \vee \bar{x}_2x_4x_5$, where r is five and $|K_n|$ is three. This node will be split into at most three nodes. First it is split on the binate variable, x_2, to get $n_1 = \bar{x}_1x_4 \vee x_2x_3$ and $n_2 = \bar{x}_2x_4x_5$. Because n_1 and n_2 are unate nodes, we stop splitting. Thus, n is represented as $n = n_1 \vee n_2$, which is a threshold function with weight-threshold vector $\langle 1, 1; 1\rangle$. Threshold network synthesis proceeds recursively by processing each of the split nodes.

```
Require: binate node $n$, $r > 0$
    $k \leftarrow (|K_n| \le r)\ ?\ K_n : r$ // pick the smaller one
    $\vec{v} \leftarrow n$ // split on binate variables when needed
3: while $|\vec{v}| \neq k \wedge \exists p \in \vec{v}$, s.t. $p$ has binate variable $x_i$ do
      $n_1 \leftarrow p(x_i, \dots)$
      $n_2 \leftarrow p(\bar{x}_i, \dots)$ // assign the resulting split nodes to vector
6:    $\vec{v} \leftarrow \{\vec{v} - p\} \vee \{n_1, n_2\}$
      if $\exists p \in \vec{v}$, s.t. $p$ is a unate node then
         break
9: return $\vec{v}$ // array of split nodes
```

Fig. 9. The binate node splitting algorithm.



```
sis>thprint

{f} = 1; [217] = 1; [218]=1;
{f} = [217] + [218];

[217] = 2; a = 1; b = 1;
[217] = a b;

[218] = 1; c = 1; d = -1;
[218] = c d̄;

sis>
```

```
sis>thprint_stats

# of PI:        4
# of PO:        1
# of levels:    3
# of nodes:     3
area:           9
network name: exam

sis>
```

```
sis>thsim 0 0 1 1

Output: 1
Next state:

sis>thsim 1 0 1 0

Output: 0
Next state:

sis>
```

Fig. 10. The framework of the threshold logic synthesis tool: TELS.

F. Complexity Analysis

In this subsection, we perform the complexity analysis of the algorithms in our threshold network synthesis methodology. The complexity of the node collapsing algorithm is $O(1)$, because the total number of operations performed by the algorithm is proportional to the fanin restriction, which is constant. The complexity of both the unate and binate node splitting algorithms is $O(|F_n| \cdot |K_n|)$. Even though ILP is NP-complete, the ILP instances being solved are small because the number of inputs of the threshold gate being synthesized cannot exceed the fanin restriction. Even then, if the ILP solver cannot find the solution in a reasonable amount of time, it declares the problem as infeasible. If that happens, the splitting algorithms in our methodology create smaller problems for the ILP solver to solve. In this way, the threshold logic synthesis problem can be solved efficiently in practice with our methodology.

G. Implementation

We implemented the proposed methodology in a tool called ThrEshold Logic Synthesizer (TELS) which has been integrated within SIS. This is the first multi-output multi-level threshold network synthesis tool to the best of our knowledge. The package currently consists of approximately $4,000$ lines of C code. The framework of TELS is shown in Fig. 10.

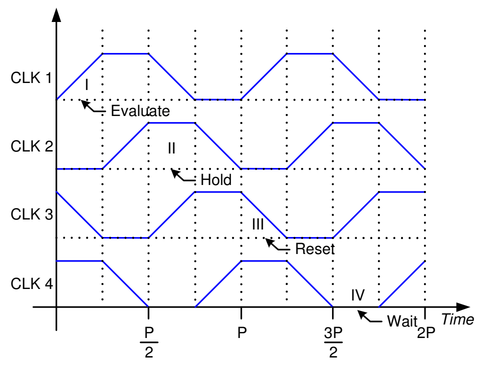


Fig. 11. Four-phase clocking for MOBILE circuits.

H. Technology Mapping

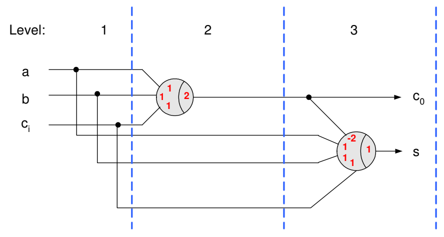
Once a threshold network has been synthesized by TELS, it can be mapped onto a target nanoscale device that is capable of implementing threshold logic. In this work, MOBILEs, which were introduced in Section I, are the target device.

A MOBILE is a self-latching RTD-HFET threshold gate, because its output is valid only when the clock is high. The MOBILE clock has four phases as shown in Fig. 11. During the *evaluation* phase, the output of a gate is computed. In the *hold* (*i.e.*, self-latching) phase, the result is valid. In the *reset* phase, the load capacitance is discharged and the gate returns to the monostable mode of operation. Finally, in the *wait* phase, the inputs of the gate are loaded with the results obtained from the predecessor gate.

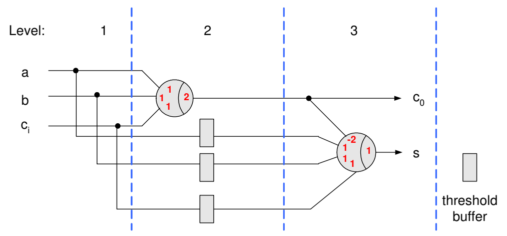
In order to make the MOBILE-based threshold network function correctly under four-phase clocking, we have to make sure that all the input signals of any embedded threshold gate arrive in the same clock phase. We have implemented this by inserting threshold buffers, wherever needed, in the network. Suppose all primary input signals arrive in the same clock phase. We examine the structure of the threshold network from the primary inputs to primary outputs. If a node fans out to several nodes and those fanout nodes are not at the same level of the network, we insert buffers to make sure that all the input signals of a node arrive in the same clock phase. An example is given in Fig. 12. The network in Fig. 12(a) implements a full adder, with logic functions $s = \bar{c}_o a \vee \bar{c}_o b \vee \bar{c}_o c_i \vee abc_i$ and $c_o = ab \vee ac_i \vee bc_i$. We observe that inputs a, b, c_i and c_o of node s do not arrive in the same clock phase. After we insert three buffers into the network, as shown in Fig. 12(b), all input signals of each node in this network arrive in the same clock phase, thus making the network function correctly. A MOBILE implementation of a threshold buffer and its symbol are shown in Fig. 12(c).

VI. Experimental Results

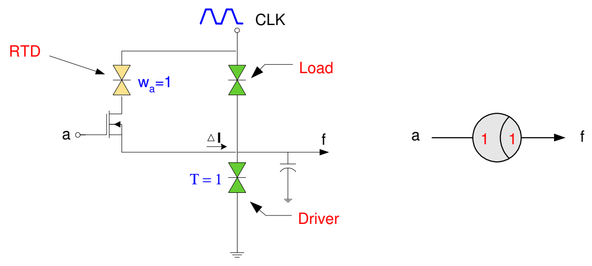
We present our experimental results in this section. The experiments were conducted on a 2.4 GHz Dell PowerEdge Pentium IV machine with 768MB RAM running Redhat Linux 8.0. We ran examples belonging to the MCNC benchmark suite through TELS. All the synthesized networks were simulated for functional correctness to validate our methodology.



(a) Network structure before inserting buffers.



(b) Network structure after inserting buffers.



(c) A MOBILE implementation of a threshold buffer and its symbol.

Fig. 12. Example of threshold network technology mapping.

A. Threshold Gate Count and Interconnect

Table I lists the results obtained by TELS for the 56 benchmarks, with the fanin restriction of a gate set to six. In this table, one-to-one mapping refers to converting each primitive gate in the algebraically-factored network into a threshold gate (the inverters, if any, are absorbed into the corresponding threshold gate by changing the weight and threshold, as was pointed out before). The threshold network synthesis results were obtained by TELS. The number of interconnects in the network was calculated by summing up the number of fanins of all the threshold gates in the network. $\%R1$ and $\%R2$ represent the gate count reduction and interconnect count reduction in the threshold network obtained by TELS compared to the threshold network obtained by one-to-one mapping, respectively.

TELS takes less than one second to synthesize most of the benchmarks. A few large benchmarks need about three to four minutes for synthesis. Comparing the results, we can see that up to 80.0% gate count reduction and 70.6% interconnect count reduction is possible, with the average being 22.7% and 12.6%, respectively.

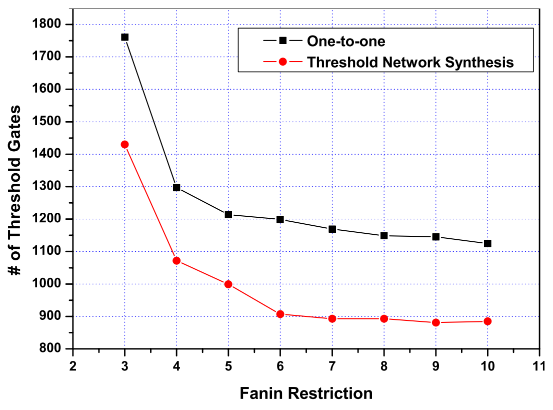


Fig. 13. Fanin-trend graph for the *pair* benchmark.

B. Trend of Threshold Gate Count with Change of Fanin Restriction

Fig. 13 demonstrates the trend in the total number of threshold gates required for the *pair* benchmark as the maximum fanin restriction is relaxed from three to ten. The gate count reduction for the one-to-one mapping case is larger than the gate count reduction for threshold synthesis case as the fanin restriction is relaxed from six to ten. This is because with a larger allowed fanin, it is possible to obtain a Boolean network with fewer gates, in general. However, there is no significant reduction in the number of threshold gates for TELS. This is understandable because as the allowed fanin increases, the likelihood of a function being threshold decreases. As reported in [16], all positive unate functions of three or fewer variables are threshold functions. However, 17 out of 20 and only 92 out of 168 positive unate functions of four and five variables, respectively, are threshold functions, not considering variable permutations. Thus, we can see that with increasing allowed fanin, the percentage of functions that are threshold decreases drastically.

C. Distribution of Weights and Thresholds

We performed experiments to see how the values of the weights and thresholds of the threshold gates in the networks synthesized by TELS are distributed. If there exist some weights or thresholds whose absolute values are much larger than others, the threshold network will not be well-balanced. We would like to avoid this phenomenon, because it may result in manufacturing problems. Fig. 14 shows that all the weights and threshold of the *dalu* benchmark assume reasonably small integer values. Another fact worth noticing is that more weights and threshold take positive values than negative values. This is because there are more positive unate variables than negative unate variables in the synthesized threshold network.

D. Parametric Variations in Weights

We also performed experiments to gauge the impact of parametric variations in the input weights on circuit functionality. We varied δ_{on} from zero to two while keeping δ_{off} fixed at one. The disturbed value w' was computed as $w' = w + v \times U(-0.5, 0.5)$, where w is the original value, v is

TABLE I
THRESHOLD NETWORK SYNTHESIS RESULTS FOR FANIN RESTRICTION OF SIX

	One-to-one mapping					Threshold network synthesis				
Benchmark	**Inputs**	**Outputs**	**Gates**	**Levels**	**Interconnect**	**Gates**	**Levels**	**Interconnect**	**%R1**	**%R2**
b1	3	4	10	4	18	8	3	16	20.0	11.1
cm42a	4	10	13	3	34	13	3	34	0	0
decod	5	16	24	3	52	24	3	52	0	0
cm82a	5	3	18	5	50	12	4	38	33.3	24.0
majority	5	1	5	3	17	1	2	5	80.0	70.6
parity	16	1	45	9	90	45	9	90	0	0
z4ml	7	4	39	8	110	19	5	64	51.3	41.8
f51m	8	8	101	8	296	82	8	266	18.8	10.1
9symml	9	1	141	10	446	110	9	412	22.0	7.6
alu2	10	6	253	27	795	197	25	715	22.1	10.1
x2	10	7	20	5	73	15	4	67	25.0	8.2
cm152a	11	1	13	4	44	11	4	42	15.4	4.6
cm85a	11	3	26	5	108	14	5	72	46.2	33.3
cm151a	12	2	14	6	47	12	5	45	14.3	4.3
alu4	14	8	517	28	1559	410	23	1407	20.7	9.8
cm162a	14	5	39	7	109	26	8	88	33.3	19.3
cu	14	11	31	6	87	24	4	76	22.6	12.7
cm163a	16	5	40	6	107	25	6	84	37.5	21.5
cmb	16	4	33	7	81	27	6	71	18.2	12.4
pm1	16	13	25	4	80	23	4	76	8.0	5.0
tcon	17	16	32	3	56	32	3	56	0	0
pcle	19	9	42	6	122	35	6	109	16.7	10.7
sct	19	15	54	6	140	38	5	115	29.6	17.9
cc	21	20	49	6	122	35	6	91	28.6	25.4
cm150a	21	1	25	5	81	21	4	77	16.0	4.9
cordic	23	2	61	9	171	49	7	155	19.7	9.4
ttt2	24	21	127	7	376	100	6	327	21.3	13.0
i1	25	16	27	5	68	23	5	63	14.8	7.4
lal	26	19	67	7	186	54	7	168	19.4	9.7
pcler8	27	17	50	7	146	47	7	143	6.0	2.1
frg1	28	3	97	12	293	59	9	233	39.2	20.5
c8	28	18	109	8	281	85	7	228	22.0	18.9
comp	32	3	89	9	315	83	8	311	6.7	1.3
my_adder	33	17	160	34	416	96	18	304	40.0	26.9
term1	34	10	278	11	761	226	10	683	18.7	10.3
count	35	16	91	12	261	79	12	241	13.2	7.7
unreg	36	16	66	4	150	50	5	134	24.2	10.7
cht	47	36	119	5	240	82	5	202	31.1	15.8
apex7	49	37	171	10	442	118	9	364	31.0	17.7
x1	51	35	293	8	866	203	7	731	30.7	15.6
dalu	75	16	1159	24	3265	810	23	2579	30.1	21.0
example2	85	66	226	9	551	182	8	489	19.5	11.3
i9	88	63	341	9	890	275	8	817	19.4	8.2
x4	94	71	264	7	696	189	8	562	28.4	19.3
i3	132	6	170	6	484	158	6	464	7.1	4.1
i5	133	66	132	19	264	66	6	260	50.0	1.5
i8	133	81	681	11	1915	570	10	1790	16.3	6.5
apex6	135	99	543	12	1348	396	12	1169	27.1	13.3
x3	135	99	660	9	1878	441	7	1514	33.2	19.4
i6	138	67	277	5	659	276	5	658	0.4	0.2
pair	173	137	1199	14	3438	907	12	2966	24.4	13.7
i4	192	6	98	6	360	74	5	336	24.5	6.7
i7	199	67	340	5	849	304	5	813	10.6	4.2
i2	201	1	244	7	766	198	7	694	18.9	9.4
des	256	245	2443	15	5743	1920	16	5180	21.4	9.8
i10	257	224	2282	39	6202	1817	35	5893	20.4	5.0

the variation multiplier and $U(-0.5, 0.5)$ is a random variable uniformly distributed between -0.5 and 0.5. A circuit fails if there exists any input vector for which the network synthesized by TELS generates a wrong output value under the disturbed weights during simulation. The failure fraction is defined as the percentage of benchmarks that failed to pass simulation. The results shown in Fig. 15 demonstrate that as δ_{on} increases, the failure rate decreases. This is because the network is more robust.

VII. CONCLUSIONS

In this paper, we introduced the first comprehensive threshold network synthesis methodology for multi-output multi-level threshold networks starting from Boolean descriptions. The algorithm in our methodology is recursive in nature and is based upon efficient heuristics that partition a logic function, if it is determined to be non-threshold, using an ILP formulation. Any fanout that occurs in the initial network is preserved in the threshold network. We have implemented the methodology on top of an existing logic synthesis tool, and validated it with

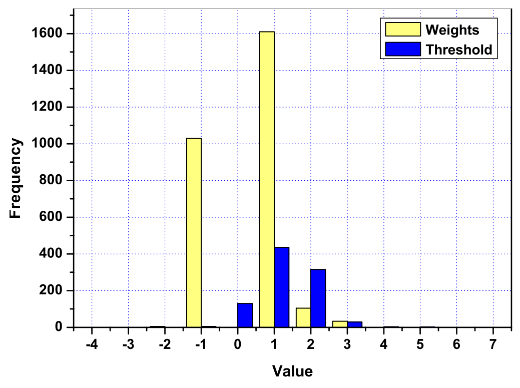


Fig. 14. Weights and threshold distribution graph for the *dalu* benchmark with fanin restriction of four. Both weights and threshold take integer values.

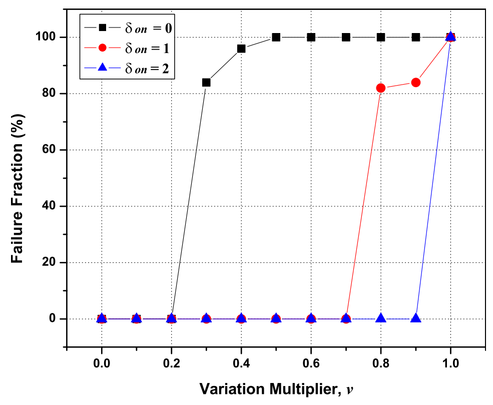


Fig. 15. The failure rate due to variations in the input weights.

a large number of benchmarks. Experimental results for the benchmarks show that the quality of the generated networks, in terms of total gate count, number of levels, and interconnect count is quite good.

Because this is the first tool for threshold network synthesis, there is room for improvement. For example, our method performs a backward traversal of the network from the outputs to the inputs. This makes the synthesized network somewhat dependent on the original network structure. Perhaps other approaches, such as divide-and-conquer, could also be used in threshold network synthesis. There may also exist better partitioning heuristics that might generate better results. Furthermore, perhaps different heuristics are required depending upon the optimization criteria. We hope that others will join in our efforts to improve this work. We also hope that integrating our methodology in commercial design automation tools will help pave the way for a smoother transition towards logic design using nanotechnologies.

References

[1] “Semiconductor Industries Association Roadmap.” http://public.itrs.net
[2] D. Goldhaber-Gordon *et al.*, “Overview of nanoelectronic devices,” *Proc. IEEE*, vol. 85, no. 4, pp. 521–540, Apr. 1997.
[3] R. Waser, *Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices*. Weinheim, Germany: Wiley-VCH, 2003.
[4] C. Pacha *et al.*, “Resonant tunneling device logic circuits,” University of Dortmund and Gerhard-Mercator University of Duisburg, Tech. Rep., July 1999.
[5] ——, “Resonant tunneling device logic: A circuit designer’s perspective,” in *Proc. European Conference on Circuit Theory & Design*, Aug. 2001.
[6] W. Prost *et al.*, “Manufacturability and robust design of nanoelectronic logic circuits based on resonant tunnelling diodes,” *Int. J. Circ. Theory Appl.*, vol. 28, no. 6, pp. 537–552, Nov. 2000.
[7] C. R. Lageweg, S. D. Cotofana, and S. Vassiliadis, “A linear threshold gate implementation in single electron technology,” in *Proc. IEEE Computer Society Wksp. VLSI*, Apr. 2001, pp. 93–98.
[8] 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 Electron Device Lett.*, vol. 17, no. 3, pp. 127–129, Mar. 1996.
[9] K. Maezawa *et al.*, “High-speed and low-power operation of a resonant tunneling logic gate MOBILE,” *IEEE Electron Device Lett.*, vol. 19, no. 3, pp. 80–82, Mar. 1998.
[10] C. Lageweg, S. Cotofana, and S. Vassiliadis, “Single electron encoded latches and flip-flops,” *IEEE Trans. Nano.*, vol. 3, no. 2, pp. 237–248, June 2004.
[11] H. Iwamura, M. Akazawa, and Y. Amemiya, “Single-electron majority logic circuits,” *IEICE Trans. Electron.*, vol. E-81C, pp. 42–48, Jan. 1998.
[12] J. P. Sun *et al.*, “Resonant tunneling diodes: Models and properties,” *Proc. IEEE*, vol. 86, no. 4, pp. 641–661, Apr. 1998.
[13] P. Mazumder *et al.*, “Digital circuit applications of resonant tunneling devices,” *Proc. IEEE*, vol. 86, no. 4, pp. 664–686, Apr. 1998.
[14] R. H. Mathews *et al.*, “A new RTD-FET logic family,” *Proc. IEEE*, vol. 87, no. 4, pp. 596–605, Apr. 1999.
[15] V. Beiu, J. M. Quintana, and M. J. Avedillo, “VLSI implementations of threshold logic - A comprehensive survey,” *IEEE Trans. Neural Networks*, vol. 14, pp. 1217–1243, Sept. 2003.
[16] S. Muroga, *Threshold Logic and its Applications*. New York, NY: John Wiley, 1971.
[17] Z. Kohavi, *Switching and Finite Automata Theory*. New York, NY: McGraw-Hill, 1978.
[18] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer Academic Publishers, 1998.
[19] R. O. Winder, “Threshold logic,” Ph.D. dissertation, Princeton University, 1962.
[20] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*. Cambridge, MA: The M.I.T. Press, 1965.
[21] G. E. Sobelman and K. Fant, “CMOS circuit design of threshold gates with hysteresis,” in *Proc. Int. Conf. Circuits & Systems*, May 1998, pp. 61–64.
[22] M. J. Avedillo *et al.*, “Low-power CMOS threshold-logic gate,” *Electronics Letters*, vol. 31, pp. 2157–2159, Dec. 1995.
[23] P. Celinski *et al.*, “Low power, high speed, charge recycling CMOS threshold logic gate,” *Electronics Letters*, vol. 37, pp. 1067–1069, Aug. 2001.
[24] V. Beiu, “Ultra-fast noise immune CMOS threshold gates,” in *Proc. Midwest Symp. Circuits & Systems*, Aug. 2000, pp. 1310–1313.
[25] C. L. Lee and C. W. Jen, “CMOS threshold gates and networks for order statistics filtering,” *IEEE Trans. Circuits Syst. I*, vol. 41, pp. 453–456, June 1994.
[26] M. J. Avedillo, J. M. Quintana, and A. Rueda, “Threshold logic,” *Wiley Encyclopedia of Electrical and Electronics Engineering*, vol. 22, pp. 178–190, 1999.
[27] M. J. Avedillo *et al.*, “Multi-threshold threshold logic circuit design using resonant tunneling devices,” *Electronics Letters*, vol. 39, pp. 1502–1504, Oct. 2003.
[28] A. L. Oliveira and A. Sangiovanni-Vincentelli, “LSAT - An algorithm for the synthesis of two level threshold gate networks,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 130–133.
[29] M. Perkowski and A. Mishchenko, “Logic synthesis for regular fabric realized in quantum dot cellular automata,” *submitted to Int. J. Multiple-valued Logic & Soft Computing*, 2004.
[30] T. Sasao, *Switching Theory for Logic Synthesis*. Norwell, MA: Kluwer Academic Publishers, 1999.
[31] E. M. Sentovich *et al.*, “Sequential circuit design using synthesis and optimization,” in *Proc. Int. Conf. Computer Design*, Oct. 1992, pp. 328–333.