Synthesis and Optimization of Threshold Logic Networks with Application to Nanotechnologies

Rui Zhang, Pallav Gupta, Lin Zhong, and Niraj K. Jha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{rzhang|pgupta|lzhong|jha}@ee.princeton.edu

***Abstract*— We propose an algorithm for efficient threshold network synthesis of arbitrary multi-output Boolean functions. The main purpose of this work is to bridge the wide gap that currently exists between research on the development of nanoscale devices and research on the development of synthesis methodologies to generate optimized networks utilizing these devices. Many nanotechnologies, such as resonant tunneling diodes (RTD) and quantum cellular automata (QCA), are capable of implementing threshold logic. While functionally correct threshold gates 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 about 60 multi-output benchmarks were performed, though the results of only 10 of them are reported in this paper because of space restrictions. They indicate that up to 77% reduction in gate count is possible when utilizing threshold logic, with an average reduction being 52%, compared to traditional logic synthesis. Furthermore, the synthesized networks are well-balanced, and hence delay-optimized.**

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 the material science and device community have enabled the creation of nanoscale devices (RTDs, QCA, to name a few) that have novel structures and properties. While CMOS is used to implement Boolean logic, many nanoscale devices implement threshold logic.

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. Today, nanotechnologies are in their infancy and the development of computer-aided design methodologies for these nanotechnologies is crucial if any of them is to replace or augment CMOS. Among existing nanoscale devices [2], RTDs and QCA are two promising nanotechnologies that are of particular interest to us because they implement threshold logic.

In this paper, we present the first comprehensive methodology for multi-level threshold logic synthesis and optimization. Once a threshold network has been synthesized, it can be mapped onto a specific target nanotechnology. The algorithm in our methodology takes into account defect tolerances in the input weights, and the fanin restriction on a threshold gate. Taking these parameters into account improves the robustness of the synthesized network. Node sharing (*i.e.*, a 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 area and delay optimized. The novel contributions of this paper 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. Section II presents background material and previous work. Section III presents an example to motivate the need for our threshold network synthesis methodology. Section IV presents some theorems and their proofs on the properties of threshold logic. Section V describes our synthesis methodology and its implementation in a tool in detail. We present our experimental results in Section VI and conclude the paper in Section VII.

Acknowledgments: This work was supported in part by NSF under grant No. CCR-0303789.

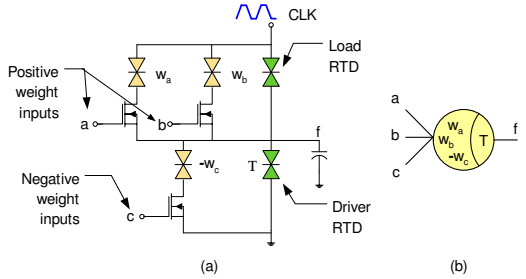


Fig. 1. A monostable-bistable logic element and its equivalent LTG.

II. Background and Previous Work

In this section, we describe some preliminary concepts. Specifically, we describe what a threshold function is and its relationship to unateness. We also review algebraically-factored and Boolean-factored expressions, and linear programming. Previous work on threshold network synthesis is also presented.

A. Threshold Logic

A linear threshold function f is a multi-input function in which each input, $x_i \in \{0,1\}$, $i \in \{1,2,\ldots,l\}$, is assigned a weight w_i such that f assumes the value 1 when the weighted sum of its inputs equals or exceeds the value of the function's threshold, T [3]. That is,

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

Parameters δ_{on} and δ_{off} represent defect tolerances that must be considered since variations (due to manufacturing defects, temperature changes, etc.) in the weights can lead to network malfunction. In this paper, we assume δ_{on} is zero and δ_{off} is one. In many past works, δ_{on} and δ_{off} are both assumed to be zero, thus providing no defect tolerance.

A linear threshold gate (LTG) is a multi-terminal device that implements a threshold function. An LTG can be considered a generalization of a conventional logic gate. An l-input NAND and an l-input NOR gate can both be realized by a single LTG. Because both gates are functionally complete, any Boolean logic function can be realized by LTGs. However, not all functions can be realized by a single LTG. A function that can be realized by a single LTG is called a *threshold function*. A network of threshold gates is called a *threshold network*.

An LTG based on RTDs and heterostructure field-effect transistors (HFET) is shown in Fig. 1(a). It outputs a logic 1 when $aw_a + bw_b - cw_c \geq T$, else a logic 0. It is called a monostable-bistable logic element [4] and its equivalent LTG is shown in Fig. 1(b).

B. Unateness

A logic function, $f(x_1,x_2,\ldots,x_l)$, 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. Unateness is an important property for threshold logic because every threshold function is unate [5] (the converse is not true, however).

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* [6]. 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, is algebraic [6]. 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 $\mathbb{C}\mathbb{X}$ is minimized subject to the p constraints given by $\mathbb{A}\mathbb{X} \leq \mathbb{B}$. Integer linear programming (ILP) [3] is a special case of linear programming that requires all of the elements in $\mathbb{X}$ to assume integer values.

E. Previous Work

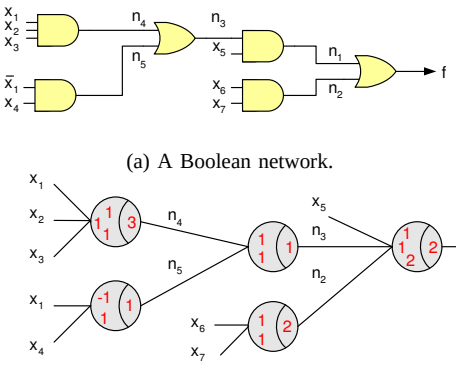
Research in threshold logic synthesis was done mostly in the 1950s and 1960s. In [7], [8], approximation methods are used to determine the input weights and threshold of a threshold function. In [5], admissible patterns on a Karnaugh map are used to determine whether a function is threshold or not. Unfortunately, because of their computational complexity, these methods are restricted to 10 or fewer variables. Linear programming and tabulation methods have been used in [3] to determine if a function is threshold or not. A CMOS implementation of threshold gates can be found in [9]. A survey of VLSI implementations of threshold logic can be found in [10]. In [11], a branch-and-bound algorithm is used to synthesize two-level threshold networks. Multi-level threshold network synthesis did not receive much attention in the 1950s and 1960s because efficient algorithms to factorize a multi-level network were unknown at that time. Most methods performed one-to-one mapping by replacing each Boolean gate in the network with a threshold gate. However, various algorithms exist today to compute the kernels and co-kernels of a network which can be used to perform algebraic or Boolean factorization [6]. In addition, methods have also been developed for Boolean network simplification. Finally, tools such as SIS [12] are available for network factorization and optimization.

III. Motivational Example

A small example is presented in this section to motivate the need for our threshold network synthesis methodology. Consider the Boolean network shown in Fig. 2(a) which has seven gates and five levels (including the inverter). If we simply replace each gate with a threshold gate, the resulting network will contain seven threshold gates and five levels. This is a sub-optimal network because some nodes in Fig. 2(a) can be collapsed into a single threshold node. However, choosing which node to collapse presents a problem. 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. In this case, it turns out that f is not a threshold function. Consequently, we must split f into smaller nodes using efficient heuristics. We choose to split f as $f = n_3x_5 \vee n_2$ where $n_2 = x_6x_7$. We synthesize n_3 next. After collapsing, n_3 is 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 $n_5 = \bar{x}_1x_4$. These three nodes are threshold functions. The synthesized threshold network is shown in Fig. 2(b). It can be seen that the number of gates and levels has been reduced by 28.6% (seven to five) and 40% (five to three), respectively.

The above example demonstrates that a threshold network synthesis methodology should address the following key issues:

- It must be able to collapse the function of a node.
- It must be able to determine if a function is threshold or not.



(a) A Boolean network.

(b) The equivalent synthesized threshold network.

Fig. 2. An example to motivate the need for our threshold network synthesis methodology.

- If a function is not threshold, it must be able to split the function into smaller functions using efficient heuristics.
- If there exist fanout nodes in the original Boolean network, it must preserve these nodes in the synthesized threshold network.

How a non-threshold function is split dictates the quality of the synthesized network. Furthermore, node sharing helps to prevent sub-network duplication during threshold network synthesis. It also helps maintain the original network structure somewhat.

IV. Theorems on Threshold Logic

We present two new theorems that describe properties of threshold logic in this section. We utilize these theorems 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 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. For simplicity, we assume $\delta_{on} = \delta_{off} = 0$. Assuming f is a threshold function, we have,

$$\sum_{k=1}^{l} w_k x_k \geq T \Rightarrow f = 1, \tag{2}$$

$$\sum_{k=1}^{l} w_k x_k < T \Rightarrow f = 0. \tag{3}$$

Note that Equations (2) and (3) represent 2^l inequalities for all 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:

$$\sum_{k=1,k\neq i}^{l} w_k x_k + w_i \bar{x}_j \geq T \Rightarrow g = 1, \tag{4}$$

$$\sum_{k=1,k\neq i}^{l} w_k x_k + w_i \bar{x}_j < T \Rightarrow g = 0. \tag{5}$$

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

$$\sum_{k=1,k\neq i,j}^{l} w_k x_k + (w_j - w_i)x_j \geq T - w_i \Rightarrow g = 1, \tag{6}$$

$$\sum_{k=1,k\neq i,j}^{l} w_k x_k + (w_j - w_i)x_j < T - w_i \Rightarrow g = 0. \tag{7}$$

If assignments for w_i and T exist such that the inequalities in Equations (2) and (3) are satisfied, then the inequalities in Equations (6) and (7) 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 weights corresponding to the variable sequence $\langle x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_l\rangle$. 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 [3]. That is, given a positive unate 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 $f(x_1, x_2, \ldots, x_l)$ 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. 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 $f(x_1, x_2) = x_1\bar{x}_2$. First, we represent f in positive unate form as $g(x_1, y_2) = x_1y_2$ where $y_2 = \bar{x}_2$. Since g is a threshold function with weight-threshold vector $\langle 1, 1; 2\rangle$, $h(x_1, y_2, x_3) = g(x_1, y_2) \vee x_3 = x_1y_2 \vee x_3$ 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 present our multi-level threshold network synthesis methodology and its implementation in this section. Fig. 3 gives a high-level overview of the main steps that comprise our methodology. The input to our methodology is an algebraically-factored multi-output combinational network, G, and its output is a functionally equivalent threshold network, G_T. An algebraically-factored circuit is used as an input because its nodes are more likely to be unate and hence possibly threshold functions. The weights and threshold are specified for each node in the synthesized network. The user can specify the fanin restriction and the defect tolerances in the weights in a threshold gate that are used during threshold network synthesis.

The synthesis algorithm begins by processing each primary output of network G. First, the node representing a primary output is collapsed. If the node represents a binate function, it is split into multiple smaller 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 smaller nodes which are then processed recursively. The synthesis algorithm terminates when all the nodes in network G are mapped into threshold nodes. We describe each step in detail in the following subsections.

The variables that are used in our algorithms are defined as follows:

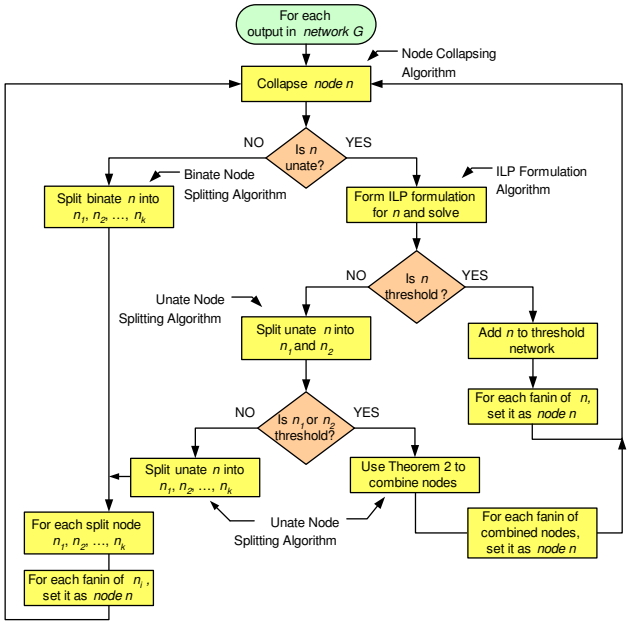


Fig. 3. Flow diagram providing a high-level overview of our threshold network synthesis methodology.

```
Require: node n, ψ > 0
    n' ← n
    l ← |F_n'|
 3: if ∀x in F_n', x ∉ P then
      while l ≤ ψ do
        for all x in F_n' do
 6:       // if neither primary input nor fanout node
          if f ∉ P ∧ x ∉ S then
            substitute the function of x into n'
 9:         l ← |F_n'|  // update l
            if l > ψ then
              n' ← undo the substitution of x into n'
12:           break
        // if all fanins are primary inputs or fanout nodes
        if ∀x in F_n', x ∈ (P ∪ S) then
15:       break
    return n' // collapsed node
```

Fig. 4. The node collapsing algorithm.

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.
ψ	Fanin restriction on a threshold gate.
l	Cardinality of F_n.
Z	Set of threshold functions.
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. Node Collapsing

The node collapsing algorithm is shown in Fig. 4. Given a node n, 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 ψ.

Note that the algorithm guarantees that the fanin of a node never exceeds ψ by choosing to undo the effects of node collapsing that was done in line 8 of the node collapsing algorithm.

To demonstrate node collapsing, let us consider the network with output node f shown in Fig. 5. Here, ψ is set to four, $l = 2$, $F_f = \{n_1, n_2\}$, $P = \{x_1, x_2, x_3, x_4\}$, $S = \{n_3\}$, and $f = n_1 \vee n_2$. Since the inputs to f are not primary inputs and l is less than

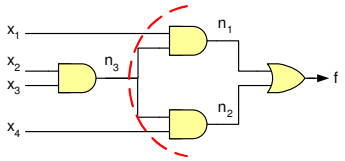


Fig. 5. Example network to demonstrate node collapsing on output f.

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

Fig. 6. The ILP formulation algorithm.

ψ, we first collapse n_1 to get $f = x_1 n_3 \vee n_2$. Now, $l = 3$ and $F_f = \{x_1, n_2, n_3\}$. Since l is still less than ψ, we continue by collapsing n_2 to get $f = x_1 n_3 \vee n_3 x_4$. 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 process stops once a fanout node is encountered. 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.

B. 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. We solve this problem by casting it in an ILP formulation. 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. 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. The algorithm for formulating the ILP problem and determining the weight-threshold vector for a threshold function is shown in Fig. 6.

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:

$$minimize: w_1 + w_2 + w_3 + T \quad (8)$$
$$subject\ to: w_1 + w_2 \geq T + \delta_{on} \quad (9)$$
$$w_1 + w_3 \geq T + \delta_{on} \quad (10)$$
$$w_2 + w_3 < T - \delta_{off} \quad (11)$$
$$w_1 < T - \delta_{off} \quad (12)$$
$$w_i \geq 0, integer,\ i = 1, 2, 3. \quad (13)$$

Require: unate node n, $\psi > 0$
if all variables appear once **then** // condition 1
  $n = n_1 \vee n_2$ s.t. $|K_{n_1}| = |K_{n_2}|$
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
else // condition 3
  $x_i \leftarrow$ most frequently appearing variable
  $n_1 \leftarrow \sum c, x_i \in c \wedge c \in K_n$
  $n_2 \leftarrow \sum c_1, c_1 \notin n_1 \wedge c_1 \in K_n$
if $n_1 \in Z$ **then** // assuming $|K_{n_1}| \geq |K_{n_2}|$
  combine nodes according to Theorem 2
  $\vec{v} \leftarrow n_1 \vee n_2$
else if $n_2 \in Z$ **then**
  combine nodes according to Theorem 2
  $\vec{v} \leftarrow n_1 \vee n_2$
else
  $k \leftarrow (|K_n| \leq \psi)\ ?\ |K_n| : \psi$ // pick the smaller one
  $\vec{v} \leftarrow \sum_{i=1}^{k} n_i$ // split n into k smaller nodes
return $\vec{v}$ // array of split nodes

Fig. 7. The unate node splitting algorithm.

The objective function for this ILP problem is defined as the summation of the weights and threshold, in order to reduce threshold gate area. Since g has been transformed into a positive unate form, only 1 and don't care ($-$) will appear in its ON-set cubes (set of cubes for which the function is 1). 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 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.

To compute the OFF-set cubes (set of cubes for which the function is 0) 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 OFF-set inequalities for g are $w_2 + w_3 < T - \delta_{off}$ and $w_1 < T - \delta_{off}$, as given in Equations (11) and (12).

By requiring the variables to be integer-valued (*i.e.*, constraint (13)), this ILP problem has an optimal solution. The weight-threshold vector for g is $\langle 2, 1, 1; 3\rangle$. Using the relationship stated in Section IV, the final weight-threshold vector for f is $\langle 2, -1, -1; 1\rangle$.

C. Unate Node Splitting and Combining

If the ILP problem for node n does not have a solution, the node must be split into smaller nodes to increase the likelihood of the split nodes being threshold functions. Fig. 7 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 cubes. For example, $n = x_1x_2 \vee x_3x_4 \vee x_5x_6$ is split as $n_1 = x_1x_2 \vee x_3x_4$, $n_2 = x_5x_6$, with $n = n_1 \vee n_2$. When a variable appears in all the cubes, we split the node into two by factoring this variable out of the node. For example, for $n = x_1x_2 \vee x_1x_3x_4 \vee x_1x_5x_6$, $n_1 = x_1$ and $n_2 = x_2 \vee x_3x_4 \vee x_5x_6$, with $n = n_1n_2$. 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.

Require: binate node n, $\psi > 0$
$k \leftarrow (|K_n| \leq \psi)\ ?\ K_n : \psi$ // pick the smaller one
$\vec{v} \leftarrow n$ // split on binate variables when needed
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
  $\vec{v} \leftarrow \{\vec{v} - p\} \vee \{n_1, n_2\}$
// split on unate variables when needed
while $|\vec{v}| \neq k \wedge \exists p \in \vec{v}$, s.t. p is unate **do**
  $n_1 \leftarrow$ split unate node p
  // assign the resulting split nodes to vector
  $\vec{v} \leftarrow \{\vec{v} - p\} \vee n_1$
return $\vec{v}$ // array of split nodes

Fig. 8. The binate node splitting algorithm.

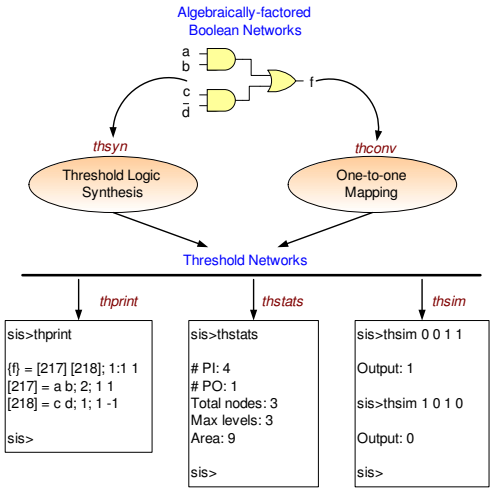


Fig. 9. The framework of the threshold network synthesis tool: TELS.

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 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 ψ and $|K_n|$. After splitting, $n = \sum_{i=1}^{k} n_i$ which is a threshold function with weight-threshold vector $\langle 1, 1, \dots, 1; 1\rangle$. The split nodes, n_i, $i \in \{1, 2, \dots, k\}$, are then processed recursively.

D. Binate Node Splitting

If a node n is binate, we split it into k smaller nodes where k is the smaller of ψ and $|K_n|$. The algorithm for binate node splitting is shown in Fig. 8. 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 use the unate node splitting algorithm, as detailed in the previous subsection.

To demonstrate binate splitting, consider $n = \bar{x}_1x_4 \vee x_2x_3 \vee \bar{x}_2x_4x_5$, where ψ is five and $|K_n|$ is three. This node is split into 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 $n_2 = \bar{x}_2x_4x_5$. Now, n_1 is further split into $n_1 = \bar{x}_1x_4$ and $n_3 = x_2x_3$. Thus, n is represented as $n = n_1 \vee n_2 \vee n_3$, which is a threshold function with weight-threshold vector $\langle 1, 1, 1; 1\rangle$. Threshold network synthesis proceeds recursively by processing each of the split nodes.

E. 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 $O(|F_n|\cdot|K_n|)$. Since ILP is NP-complete, our synthesis problem is NP-complete in theory. However, in practice, the ILP solver is implemented such that if the optimal solution cannot be found 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 network synthesis problem can be solved efficiently in practice with our methodology.

TABLE I
THRESHOLD SYNTHESIS RESULTS WITH FANIN RESTRICTION SET TO 3

Benchmark	One-to-one mapping			Threshold network synthesis		
	Gates	Levels	Area	Gates	Levels	Area
cm152a	28	4	99	13	4	69
cordic	92	9	307	39	8	219
cm85a	70	8	254	16	6	158
comp	181	12	625	70	9	435
cmb	41	7	142	16	7	103
term1	397	12	1,459	144	16	787
pm1	49	5	176	22	3	119
x1	428	10	1,589	144	10	968
i10	2,874	49	10,934	1,276	47	7,261
tcon	24	2	80	32	2	96

F. 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 3,500 lines of C code. We integrated a linear programming tool called *LP_SOLVE* [13] in SIS to solve the ILP problems. The framework of TELS is shown in Fig. 9. Currently, it supports five commands, which perform one-to-one mapping, threshold synthesis and simulation, and displaying of network information.

VI. EXPERIMENTAL RESULTS

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

A. Threshold Gate Count and Area

Due to space limitations, Table I lists the results for only 10 of the 60 benchmarks. In this table, one-to-one mapping refers to replacing each gate in the optimized Boolean network with a threshold gate. The optimized Boolean network was obtained by running the *script.boolean* script in SIS. The threshold network synthesis results were obtained by running the *script.algebraic* script and then synthesizing the threshold network from the resulting algebraically-factored network. The area of the network, A, was calculated using the following equation:

$$A = \sum_{g \in G_T} \sum_{i=1}^{l} (|w_i| + |T|)A_u, \tag{14}$$

where g is a threshold gate in network G_T, l is the number of inputs in gate g, w_i is the weight of input i, A_u is the unit area of an RTD with $w = 1$, and T is the threshold of the gate. In our case, we let A_u equal one. The HFET area is ignored because it is typically much smaller than the RTD area.

The total time required to algebraically factor and synthesize the benchmarks was less than one second in each case. On an average, 42% of the total execution time was spent on threshold network synthesis while the remaining time was spent on factoring the network. Comparing the results, we see that 52% average reduction is possible in gate count. In some cases, such as *tcon*, we do worse than a one-to-one mapping because there exist Boolean functions that require more threshold gates than Boolean gates. However, this is not a significant problem because we can always choose the better of the two networks, thereby, guaranteeing that TELS will never

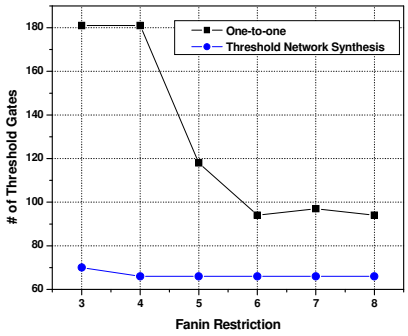


Fig. 10. Gate-count vs. fanin restriction for *comp*.

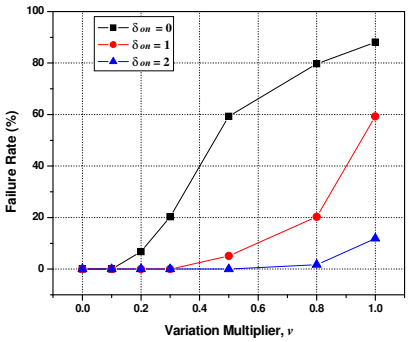


Fig. 11. The failure rate due to variations in the input weights.

output a network requiring more gates than that required for one-to-one mapping.

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

Fig. 10 demonstrates the trend in the total number of threshold gates required for the *comp* benchmark as the maximum fanin restriction is relaxed from three to eight. There is a significant difference in gate count in the one-to-one mapping case as the fanin restriction is relaxed. This is because with larger allowed fanin, it is possible to decompose a factorized network better. 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 [3], 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. The overall conclusion is that a fanin restriction of three to five gives good results.

C. Parametric Variations in Weights

We performed experiments to gauge the impact of parametric variations in the input weights on the circuit functionality. We varied δ_{on} from zero to three and δ_{off} was 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 the variation multiplier and $U(-0.5, 0.5)$ is a random variable uniformly distributed between -0.5 and 0.5. The circuit fails if there exists any input vector with which TELS generates a wrong output value under the disturbed weights during simulation. The failure rate is defined as the percentage of benchmarks that failed to pass simulation. The results shown in Fig. 11 demonstrate that as δ_{on} increases, the failure rate decreases. This is because the network is more robust. The tradeoff is that the network area increases as shown in Fig. 12 for the case $v = 0.8$.

VII. Conclusions

In this paper, we introduced the first comprehensive threshold network synthesis methodology for multi-output multi-level networks.

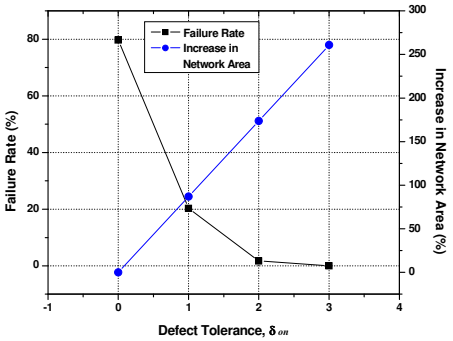


Fig. 12. The relationship between failure rate, network area, and defect tolerance due to variations in the input weights with variation multiplier $v = 0.8$.

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 logic sharing that occurs in the algebraically-factored network is reflected in the threshold network. We have implemented the methodology on top of an existing logic synthesis tool to produce the first threshold network synthesis tool, and validated it by running the tool on a large number of benchmarks and simulating the synthesized networks. Experimental results for the benchmarks show that the quality of the generated networks, in terms of total gate count and the number of levels, is very 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 upon 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] S. Muroga, *Threshold Logic and its Applications*. New York, NY: John Wiley, 1971.
[4] K. J. Chen, K. Maezawa, and M. Yammoto, "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.
[5] Z. Kohavi, *Switching and Finite Automata Theory*. New York, NY: McGraw-Hill, 1978.
[6] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA: Kluwer Academic Publishers, 1998.
[7] R. O. Winder, "Threshold logic," Ph.D. dissertation, Princeton University, 1962.
[8] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*. Cambridge, MA: The M.I.T. Press, 1965.
[9] G. E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis," in *Proc. Int. Conf. Circuits & Systems*, vol. 2, May 1998, pp. 61–64.
[10] V. Beiu, J. M. Quintana, and M. J. Avedillo, "VLSI implementations of threshold logic - A comprehensive survey," *Tutorial at Int. Joint Conf. Neural Networks*, July 2003.
[11] 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.
[12] E. M. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, Oct. 1992, pp. 328–333.
[13] M. Berkelaar, "Linear programming solver." http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml