# Decomposition Based Approach for Synthesis of Multi-Level Threshold Logic Circuits

Tejaswi Gowda and Sarma Vrudhula
Consortium of Embedded Systems, School of Computing and Informatics, Arizona State University, Tempe, AZ.
{tejaswi, vrudhula}@asu.edu

***Abstract*— Scaling is currently the most popular technique used to improve performance metrics of CMOS circuits. This cannot go on forever because the properties that are responsible for the functioning of MOSFETs no longer hold in nano dimensions. Recent research into nano devices has shown that nano devices can be an alternative to CMOS when scaling of CMOS becomes infeasible in the near future. This is motivating the need for stable and mature design automation techniques for threshold logic since it is the design abstraction used for most nano-devices. This paper presents a new decomposition theory that is based on the properties of threshold functions. The main contributions of this paper are: (1) A new method of algebraic factorization called the *min-max* factorization. (2) A decomposition theory that uses this new factorization to identify and characterize threshold functions. (3) A new threshold logic synthesis methodology that uses the decomposition theory. This synthesis methodology produces circuits that are better than the previous state of art (27% better gate count and comparable circuit depth).**

## I. Introduction

Threshold logic (TL) has long been known as an alternative way to compute Boolean functions [13], [16], [7]. Much of the earlier work on TL dates back to the 1960s, which focused primarily on exploring the theoretical aspects, with little attention being paid to the synthesis and optimization of large, multi-level TL networks. The lack of efficient implementations of TL gates, when compared to static fully complementary MOS transistor networks and the rapid development of synthesis and optimization tools for Boolean logic design led to a loss of interest in developing similar infrastructure for designing TL circuits. The situation is now changing in favor of threshold logic [17], [10], [6].

The scaling of MOSFETs that has been taking place for over three decades is expected to continue for at least another decade, after which we will reach a point where transitioning to non-CMOS technologies will be necessary [1]. A large amount of research is currently in progress to find the best alternative. A few examples of *post-CMOS* devices are resonant tunneling diodes (RTDs) [15], single electron transistors (SETs) [14], quantum cellular automata (QCA) [5] and carbon nano-tube FETs (CNT-FETs) [3]. A common and important characteristic of these devices is that they can be used to realize threshold logic very efficiently [17]. Efficient CMOS implementations of threshold gates are also currently available [6], [4]. Consequently, there has been a resurgence of interest in threshold logic and synthesis and verification methods that are applicable to large, multi-level threshold networks [17], [11], [9], [2], [10].

**Salient Features:**

**(1)** In this paper we present a new approach to the synthesis of multi-level threshold networks. **(2)** The key feature of the method is the determination of whether or not a given Boolean function is a threshold function. The traditional approach of doing this is based on determining the satisfiability of a large number of integer linear inequalities. Such a method is only practical for functions with a few inputs. **(3)** The proposed method eliminates the use of linear programming to determine weights and threshold for a threshold function. **(4)** We present a new theory based on a new factorization of an SOP of a Boolean function. We use the new factorization to identify sub-functions of a Boolean function that are threshold. This is used to *decompose* a Boolean function into its constituent threshold functions. **(5)** We apply this decomposition method to the problem of synthesis of threshold circuits.

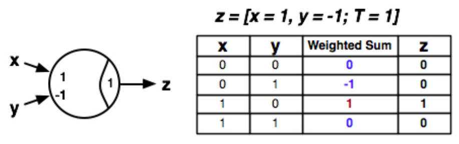
This research is supported by NSF Grant numbers 015516-001 and 019772-001 and a grant by the Arizona Department of Commerce for Center for Low Power Electronics, grant number 009058-010

## II. Background

A threshold gate has one or more binary inputs, $x_1, x_2, \ldots, x_n$, and a single binary output [7]. The gate is characterized by a set of *weights*, $W = w_1, w_2, \ldots, w_n$, where $w_i$ is the weight associated with input $x_i$, and a *threshold* $T$. The output of a threshold gate is defined as follows:

$$y = \begin{cases} 1 & \text{if} \sum_{i=0}^{n} w_i x_i \geq T \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

A Boolean function is called a threshold function if it can be implemented by a single threshold gate. Since the threshold gate realizing a function $f$ is completely characterized by the set of weights $W$ and the threshold $T$, we represent it by $f = [W; T] = [w_1, w_2, \ldots, w_n; T]$. Figure 1 shows a simple example of a threshold gate $[x = 1, y = -1; T = 1]$.



| x | y | Weighted Sum | z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | -1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

Fig. 1. Threshold gate example

Threshold functions are a subset of unate functions [13]. Since the primitive gates such as OR, AND, NOR and NAND are threshold, one can view a multi-level logic network composed of such gates as a special case of a threshold network. However, the advantage of using threshold logic is that much more complex functions can be implemented in a single threshold gate. Hence a multi-level threshold network may be viewed as a generalization of a traditional logic network using much more complex primitives. This can lead to significant reduction in gate count and, in many cases, circuit depth. These reductions translate into area, power and delay reduction. For example, the function $ab(c+d)+cd(a+b)$ can be implemented by a single threshold gate. We would need 5 Boolean *AND/OR* gates in 3 levels to implement this function. Since not all Boolean functions are threshold, an arbitrary Boolean function will have to be implemented as a multi-level threshold network.

Determining whether or not a given Boolean function is threshold is an important problem for synthesis, verification and optimization of threshold logic networks. Another important problem is to decompose a Boolean function into sub-functions that are threshold, i.e. to determine the *parts* of the Boolean function that are threshold. After a function has been identified as threshold it is necessary to assign weights and a threshold for the gate. As stated earlier, currently the task of identifying threshold functions and assigning weights is done using the *ILP formulation* [16], [17]. In this paper we present an efficient (non-ILP) method to address all three problems – identifying functions or sub-functions that are threshold, and assigning weights and thresholds. Only recently has there been a significant effort in the area of threshold synthesis [17], [11], [2]. Most of these methods use the ILP formulation to determine whether or not a function is a threshold function and for the weight-threshold assignment. Moreover, these methods use a Boolean circuit and local merging of Boolean gates to obtain a threshold circuit. Thus the quality of result depends on the circuit representation used as input.

## III. Preliminaries

### A. Definitions

*Positive Threshold Function:* A positive threshold function is one in which all the variable weights in the weight-threshold assignment are positive. A positive threshold function also a positive unate function [13]. For example $F = a + bc \equiv [a = 2, b = 1, c = 1; T = 2]$ is a positive threshold function. Note that it is also positive unate.
*Support Set:* The set of all variables on which the function depends is called the support set of the function. The support set of function $F$ is denoted by $Supp(F)$. *e.g:* The support set of $F = a + c$, $Supp(F) = \{a, c\}$.
*Don't Care Variable:* A variable $d$ is said to be a *don't care* variable of a function $F$ if and only if $F_{d=0} = F_{d=1}$. Don't care variables do not belong to the support set of a function.
*Complete Sum:* An SOP formula is a complete sum (a sum of all prime implicants and only prime implicants [12]) if and only if:

1) no term includes any other term,
2) the consensus of any two terms of the formula either does not exist or is contained in some other term.

The complete sum of function $F$ is denoted by $CS(F)$. For example, the complete sum of $ab' + ab + c$ is $a + c$.
*$\succeq$-Ordering:*
For a function $F$ if $F_{x=1,y=0} \supseteq F_{x=0,y=1}$, it is denoted as $x \succeq y$. $x \preceq y$ is defined similarly. If $F_{x=1,y=0} \supset F_{x=0,y=1}$, it is denoted as $x \succ y$. $x \prec y$ is defined similarly.

For a pair of variables $x$ and $y$, if $x \succeq y$ and $x \preceq y$, it is denoted as $x \approx y$.

For a threshold function, $w_x > w_y$ implies $x \succeq y$, and $w_x = w_y$ implies $x \approx y$ [16]. It is also known that for a threshold function $F$, $Supp(F)$ can be totally pseudo-ordered using the $\succeq$-relation. For more detals on these operators the reader is referred to [16].

### B. Min-Max Literal (MML) Factorization

Factorization of Boolean SOP is done to reduce the number of literals and thereby obtain more compact representations. Algebraic factorization is the *algebraic* division of a Boolean function. If $D$ is the divisor used to factor $F$, then $F = Q.D + R$, where $Q$ and $R$ are the quotient and remainder obtained by the algebraic division of $F$ by $D$. $Q$ and $R$ may be further factored to obtain a more compact factored form. Many different factoring techniques have been presented [12]. The main difference between these different techniques is the way in which the divisors are chosen. One factoring technique called the *best literal factorization* uses that literal for the divisor which occurs in the greatest number of cubes [12]. For example, for $F = ab + ac + de = a(b + c) + de$, $a$ is the *best literal* as it occurs in two cubes, which is more than the number of cubes in which any other literal occurs. Here $Q = b + c$, $D = a$, and $R = de$ and $F = Q.D + R$.

We propose a new kind of factorization. As we'll show later this is well suited for our decomposition procedure. We call this factorization the *min-max literal factorization*. This is very similar to the best literal factorization in that a single literal is used as a divisor, but differs slightly in the way the divisor is chosen.

Let $C_j$ represent the set of all cubes in a SOP that contain $j$ literals. The min-max literal is a literal that occurs in the greatest number of cubes in $C_k$, where $k$ is the size of the smallest cube. For example, in $ab + ce + ad + bcd$, $a$ is the min-max literal as it occurs more often in $C_2 = \{ab, ce, ad\}$, than any other literal.

In case of a tie, the literals that occur in an equal number of cubes in $C_k$ are then compared to each other using the occurrences of these literals in $C_{k+1}$ and so on, until the tie is resolved. For example, in $abc + ad + ae + de$, again $a$ is the min-max literal. Even though $a, d$ and $e$ all occur in 2 cubes in $C_2 = \{ad, ae, de\}$, the tie is broken using $C_3 = \{abc\}$, since $a$ occurs in one cube of $C_3$, whereas $d$ and $e$ are not present in any cube in $C_3$.

In case the tie is not broken even after comparing the variable occurrences in $C_l$, where $l$ is size of the largest cube, the tie is broken arbitrarily. In $ab + ac + bc$, $a$, $b$ or $c$ can be chosen as the min-max literal, as they all occur in equal number of cubes in $C_2$ and the largest cube size is 2.

### C. The Min-Max Literal Factor Tree (MMLFT)

After repeatedly factoring an SOP using the min-max literal factorization, we can represent the factored form using a factor tree. This factor tree is a binary tree which has a labeled left edge. The leaves of the tree are Boolean *AND*, Boolean *OR*, a single literal or constants 1 and 0. The left edge of any node is labeled using the min-max literal chosen to be the divisor of the function of the node. The left child represents the quotient obtained by algebraic division, and the right child the remainder (see Figure 2).

An example MMLFT for $G = a + bc + bd + be + cd + ce = a + b(c + d + e) + c(d + e)$ is shown in Figure 3.

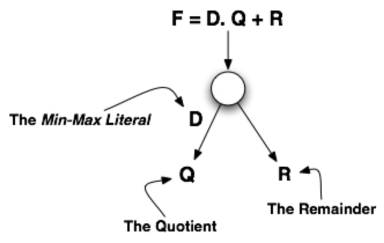


Fig. 2. The Min-Max Literal Factor Tree

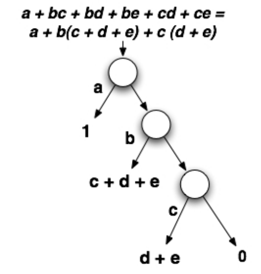


Fig. 3. Example Factor Tree

## IV. Threshold Logic Decomposition

### A. Identification of Threshold Functions : Overview

Our aim is to identify parts of a Boolean function that are threshold. The starting point of the decomposition procedure is a min-max literal factor tree that is constructed from a given SOP [8]. To make the factor tree more compact, we ensure that the SOP is minimal with respect to single cube containment. The MML factorization is crucial for the decomposition procedure. For a threshold function, the min-max literal represents the variable with the highest weight (see Lemma 6). Moreover, many of the properties that the decomposition procedure is based on are valid only for a MML factor tree.

Without loss of generality, we assume that all literals are positive. Therefore $a$ and $a'$ are considered to be two seperate literals. All weights that are assigned during the decomposition procedure are thus positive. The exact weights can be easily obtained by the use of Lemma 7 (also in [16], *pg. 58*). For example, if the decomposition procedure yields the following weight-threshold assignment: $[a' = 2, b = 1, c = 1; T = 3]$, then by Lemma 7, this is equivalent to $[a = -2, b = 1, c = 1; T = 1]$.

Figure 4 shows the flow of procedure for decomposition of a threshold function. The procedure traverses the MML factor tree in a bottom-up fashion. By construction, the leaf nodes (*AND/OR* functions, a single literal, or constants 1, 0) of the tree are trivially threshold. The core of the procedure is to determine whether or not a node $F$ is a threshold function given that its two children are threshold functions. If so, the weights are determined; otherwise, the function $F$ is transformed by reordering the MML factor tree after decomposing one child of $F$. We now explain the steps of the algorithm shown in Figure 4.

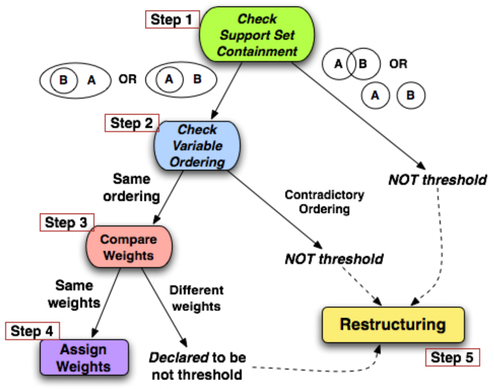


Fig. 4. The Threshold Decomposition Flow

In what follows, let $A$ and $B$ be the *left* and *right* children of a node $F$, with the left edge labeled by the literal $x$ (i.e., $F = xA+B$). Suppose $A$ and $B$ are threshold functions. Lemma 5 states that if in a MML factor tree, the support set of one child is not contained in the support set of the other child, then $F$ is not a threshold function. Therefore, step 1 in Figure 4 checks this condition. If it is not threshold, $F$ is *restructured*.

Next, if the conditions of Lemma 5 are satisfied, we then check the conditions of Lemma 4, which states that in a MML factor tree, if there exists a pair of variables in $A$ and $B$ that have different $\succeq$-ordering, then $F$ is not a threshold function. If this test succeeds, we compare the variable weights of $A$ and $B$. If they are the same, then $F$ will be a threshold function (Lemma 1) and we proceed with assigning the weights (also indicated in Lemma 1). If the weights are not the same, $F$ may or may not be a threshold function. At this point we *declare* $F$ to be a non-threshold function and proceed with the restructing step.

## B. Weight assignment for leaf nodes

In order to increase the chance of identifying threshold functions, we use the following rules to assign weight to the leaf nodes (note that we treat all literals as positive literals):

**(1)** The *AND*, *OR*, constants 1 and 0 nodes are assigned the same weights as the weights assigned to the *sister*[*] node function. **(2)** For an *OR* node the exact same weights of the sister node are assigned and the threshold is set to be equal to the smallest variable weight. This is a valid weight-threshold assignment as when *any one* of the inputs is 1, the *OR* function outputs 1 (Figure 5(a)). **(3)** For an *AND* node the exact same weights of the sister node are assigned and the threshold is set to be the sum of all weights. This is a valid weight-threshold assignment, as only when *all* of the input variables are 1, the *AND* function outputs 1 (Figure 5(b)). **(4)** The weights are similarly made identical to that of the sister node in case of a constant 1 node. The threshold is set to 0. This works as we have all our weights $> 0$ (positive threshold function) and no matter what the state of input variables are the weighted sum of inputs is always $\geq 0$, which is the threshold (Figure 5(c)). **(5)** For a constant 0 node after setting the same weights as that of the sister node, the threshold is set to be one greater than the sum of all weights (Figure 5(d)). **(6)** If the leaf node is *AND/OR* and the sister node has not been assigned weights and a threshold, we assign a weight of 1 to all inputs. The threshold is set to be 1 for an *OR* node and it is set to be the cardinality of the support set in case of an *AND* node.

[*]Two nodes are *sister* nodes if and only if they have the same parent.

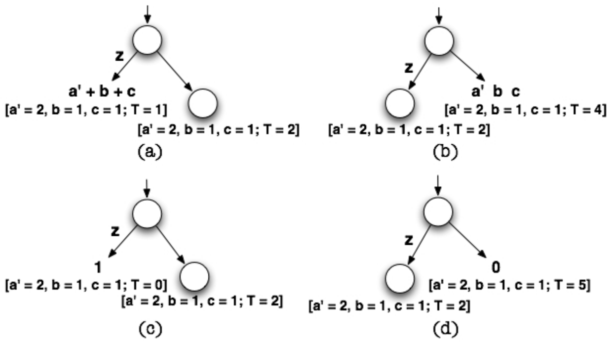

z
a' + b + c
[a' = 2, b = 1, c = 1; T = 1]
[a' = 2, b = 1, c = 1; T = 2]
(a)
z
a' b c
[a' = 2, b = 1, c = 1; T = 4]
[a' = 2, b = 1, c = 1; T = 2]
(b)
z
1
[a' = 2, b = 1, c = 1; T = 0]
[a' = 2, b = 1, c = 1; T = 2]
(c)
z
0
[a' = 2, b = 1, c = 1; T = 5]
[a' = 2, b = 1, c = 1; T = 2]
(d)

Fig. 5. Weight-threshold assignment to leaf nodes

*Weight Resynthesis:* Lemma 2 states that the don't care variables in a threshold function must have a weight strictly less than the weights of all other variables. In the course of assigning weights to the leaf nodes, this condition may be violated. Figure 6(a) shows an example of this violation. When assigning weights to the variables of the right child of node 1, the weight assignment of node 2 is copied. As a result, $e$, which is a don't care variable of the right child of node 1, violates Lemma 2 (weights of $e$ and $d$ are the same). Weight resynthesis is performed on the subtree rooted at node 2 with the constraint that variable $e$ receive a weight strictly less than the weight of all other variables. This is shown in Figure 6(b). Finally, since such an assignment is possible, node 1 is determined to be a threshold function.

## C. Weight assignment for non-leaf nodes

*Lemma 1:* If $F = xA+B$ in a MML factor tree, and $A$ and $B$ are known to be threshold, with a weight-threshold assignment $[W; T_A]$ and $[W; T_B]$ (identical input weights) respectively, and $T_B > T_A$, then $F$ is also a threshold function with $w_x = T_B - T_A$ and $F \equiv [W \cup \{T_B - T_A\}; T_B]$.

**Example:** Figure 7 shows a MML factor tree of the function $a(b + c) + bc$. The leaves are trivial threshold functions whose weight-threshold pairs can be trivially assigned. Now consider the parent node. Both children satisfy Lemma 5 and 4, and have the same weights. Hence weight-threshold assignment for the parent can be determined by Lemma 1.

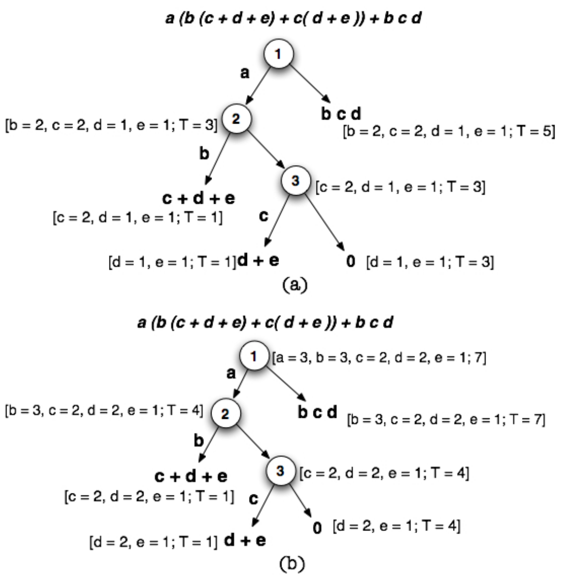

a (b (c + d + e) + c( d + e )) + b c d
[b = 2, c = 2, d = 1, e = 1; T = 3]
b c d
[b = 2, c = 2, d = 1, e = 1; T = 5]
c + d + e
[c = 2, d = 1, e = 1; T = 1]
[c = 2, d = 1, e = 1; T = 3]
[d = 1, e = 1; T = 1] d + e
0 [d = 1, e = 1; T = 3]
(a)
[a = 3, b = 3, c = 2, d = 2, e = 1; 7]
[b = 3, c = 2, d = 2, e = 1; T = 4]
[b = 3, c = 2, d = 2, e = 1; T = 7]
[c = 2, d = 2, e = 1; T = 1]
[c = 2, d = 2, e = 1; T = 4]
[d = 2, e = 1; T = 1] d + e
0 [d = 2, e = 1; T = 4]
(b)

Fig. 6. Resynthesis Example

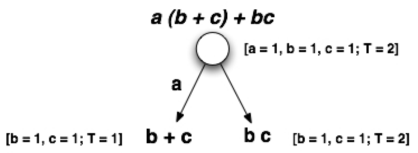

a (b + c) + bc
[a = 1, b = 1, c = 1; T = 2]
a
[b = 1, c = 1; T = 1] b + c
b c [b = 1, c = 1; T = 2]

Fig. 7. Weight Assignment Example

### D. *Restructuring the MML Factor Tree*

The restructuring of the factor tree is done when the node $F$ is declared to be a non-threshold function, even though its children are threshold. When we encounter a node that is not threshold but its children are threshold, we decompose the MML factor tree. We remove the *larger* of the two children (one with the bigger support set) and modify the tree so that we can continue to identify more threshold functions.

There are two different restructuring rules depending on which child is eliminated. This is shown in Figure 8. The function of the child with a larger support set is assigned a single literal (here $X$) and the tree is reordered. Reordering is straightforward if the left child is replaced by a literal. Reordering is more involved if the right child is replaced by a literal. Note that this reordering does not change the function represented by the tree. The procedure only helps to identify sub trees in the MML factor tree that represent threshold functions.

## V. Application to Synthesis of Threshold Circuits

A threshold circuit is a directed graph. The nodes in this graph represent threshold elements and the directed edges represent input-output interconnection between different gates. Each threshold element has a weight-threshold assignment that fully characterizes the element. The objective of the synthesis procedure is to generate a threshold circuit that implements the specified function.

The decomposition methodology developed earlier is an integral part of the synthesis procedure. Note that the decomposition procedure identifies sub functions that are threshold. If we perform repeated decomposition of the MMLFT until the root node is declared to be threshold, we get a network of threshold gates that will implement the required function. **Example:** Consider the function $ab + cd$.

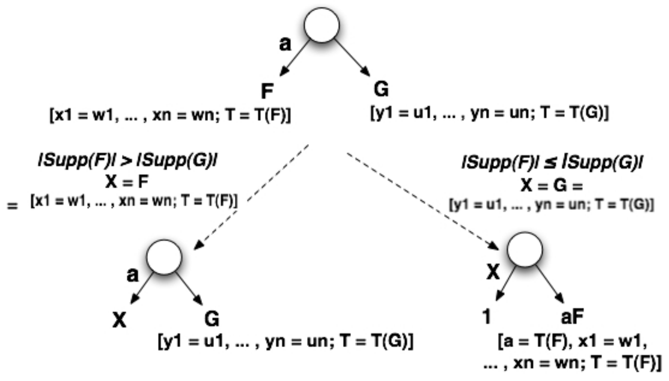

a
F
G
[x1 = w1, ... , xn = wn; T = T(F)]
[y1 = u1, ... , yn = un; T = T(G)]
|Supp(F)| > |Supp(G)|
X = F
= [x1 = w1, ... , xn = wn; T = T(F)]
|Supp(F)| ≤ |Supp(G)|
X = G =
[y1 = u1, ... , yn = un; T = T(G)]
a
X
G
[y1 = u1, ... , yn = un; T = T(G)]
X
1
aF
[a = T(F), x1 = w1, ... , xn = wn; T = T(F)]

Fig. 8. Restructuring Rules

This is a not a threshold function [17]. By using the decomposition methodology repeatedly we can obtain the threshold circuit shown in Figure 9 (b).

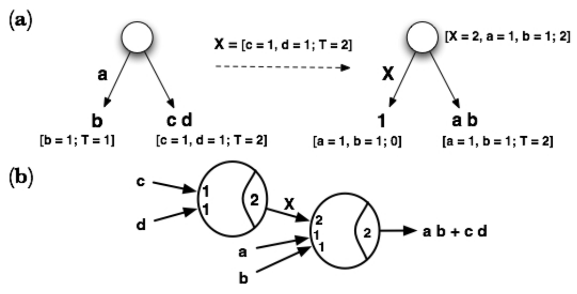


Fig. 9. Synthesis Example: (a) Synthesis procedure. (b) Synthesized circuit

*The Synthesis Flow:* To generate threshold circuits that implement multi-output functions we follow the synthesis flow shown in Figure 10. We first use SIS [8] to obtain an optimized circuit graph. Each node in the circuit graph represents a complex Boolean function. For each node the MML factor tree of the node function is constructed. Repeated decomposition is performed to obtain a threshold circuit for the function of the node. Once a threshold circuit is obtained for all nodes in the circuit graph, we would have generated the required multi-output threshold circuit.

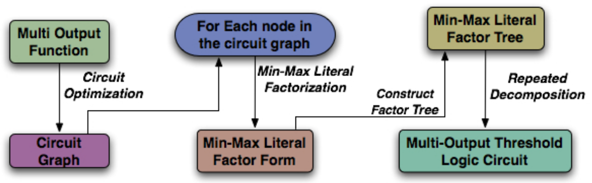

Multi Output Function
Circuit Optimization
Circuit Graph
For Each node in the circuit graph
Min-Max Literal Factorization
Min-Max Literal Factor Form
Construct Factor Tree
Min-Max Literal Factor Tree
Repeated Decomposition
Multi-Output Threshold Logic Circuit

Fig. 10. The Threshold Synthesis Flow

## VI. Experimental Results

The proposed algorithm was implemented in Python and was run on an Apple iBook G4 with 1 GB RAM. Circuits in the MCNC benchmark suite were synthesized. The results are reported in Table I. Gate count and depth are non-technology specific metrics for area and delay. We report these as they are reported in the previously published works [17], [2]. Thus we use the same to compare our approach against the state-of-art synthesis method. The table lists the gate count and levels for the benchmark circuits when implemented as Boolean circuits and as TL circuits by the method described in [17] (which has a fanin restriction of 6 input). Compared to the method in [17], our method generates circuits with comparable depth and 27% fewer gates on average (66% at best). Our method does not have a restriction on the fanin of gates, however such a restriction can be imposed if needed.

TABLE I
COMPARISON WITH PREVIOUS WORK

| Bench- | Boolean Circuit | | Method in [17] | | Proposed Method | |
|---|---|---|---|---|---|---|
| mark | Gates | Depth | Gates | Depth | Gates | Depth |
| b1 | 10 | 4 | 8 | 3 | 6 | 3 |
| cm42a | 13 | 3 | 13 | 3 | 12 | 3 |
| decod | 24 | 3 | 24 | 3 | 18 | 2 |
| cm82a | 18 | 5 | 12 | 4 | 12 | 6 |
| majority | 5 | 3 | 1 | 2 | 1 | 1 |
| parity | 45 | 9 | 45 | 9 | 30 | 8 |
| z4ml | 39 | 8 | 19 | 5 | 16 | 8 |
| f51m | 101 | 8 | 82 | 8 | 40 | 5 |
| 9symml | 141 | 10 | 110 | 9 | 82 | 9 |
| alu2 | 253 | 27 | 197 | 25 | 152 | 26 |
| x2 | 20 | 5 | 15 | 4 | 15 | 4 |
| cm152a | 13 | 4 | 11 | 4 | 8 | 4 |
| cm85a | 26 | 5 | 14 | 5 | 14 | 7 |
| cm151a | 14 | 6 | 12 | 5 | 17 | 5 |
| alu4 | 517 | 28 | 410 | 23 | 297 | 33 |
| cm162a | 39 | 7 | 26 | 8 | 18 | 7 |
| cu | 31 | 6 | 24 | 4 | 16 | 4 |
| cm163a | 40 | 6 | 25 | 6 | 17 | 6 |
| cmb | 33 | 7 | 27 | 6 | 14 | 3 |
| pm1 | 25 | 4 | 23 | 4 | 16 | 3 |
| tcon | 32 | 3 | 32 | 3 | 16 | 2 |
| pcle | 42 | 6 | 35 | 6 | 26 | 9 |
| sct | 54 | 6 | 38 | 5 | 33 | 4 |
| cc | 49 | 6 | 35 | 6 | 23 | 3 |
| cm150a | 25 | 5 | 21 | 4 | 32 | 6 |
| cordic | 61 | 9 | 49 | 7 | 52 | 6 |
| ttt2 | 127 | 7 | 100 | 6 | 84 | 6 |
| pcler8 | 50 | 7 | 47 | 7 | 34 | 9 |
| frg1 | 97 | 12 | 59 | 9 | 24 | 6 |
| c8 | 109 | 8 | 85 | 7 | 75 | 6 |
| comp | 89 | 9 | 83 | 8 | 57 | 13 |
| my_adder | 160 | 34 | 96 | 18 | 33 | 17 |
| term1 | 278 | 11 | 226 | 10 | 102 | 8 |
| count | 91 | 12 | 79 | 12 | 48 | 18 |
| unreg | 66 | 4 | 50 | 5 | 48 | 3 |
| cht | 119 | 5 | 82 | 5 | 73 | 3 |
| apex7 | 171 | 10 | 118 | 9 | 94 | 9 |
| x1 | 293 | 8 | 203 | 7 | 82 | 6 |
| example2 | 226 | 9 | 182 | 8 | 137 | 8 |
| x4 | 264 | 7 | 189 | 8 | 159 | 5 |
| apex6 | 543 | 12 | 396 | 12 | 310 | 12 |
| x3 | 660 | 9 | 441 | 7 | 415 | 8 |
| pair | 1199 | 14 | 907 | 12 | 609 | 15 |

The most important advantage of our method when compared with these two previous approaches is that it gives a combinatorial method and a theoretical underpinning for synthesizing TL circuits as oppposed to the heuristic of localized merging of Boolean gates.

The method in [2] generates feed forward TL networks in which there is only one gate in each level. This method is extremely expensive in terms of computation time. It requires more than a minute even for small circuits of only 2 TL gates. In comparison our method takes an average of 2 seconds to complete execution. Even for circuits with a large number of gates (*e.g:* the *pair* benchmark circuit which has over 600 threshold gates) it takes no more than 6 seconds to generate the threshold circuit.

To better compare the results, we use the histogram in Figure 11. The x-axis of the histogram represents the number of gates in the Boolean circuit. The y-axis of the histogram plots the average number of gates in the threshold circuits generated by the two methods to implement the same benchmarks. *Example:* Consider the circuits that needed 200-500 gates in the Boolean implementation. An average of 200 gates were needed by the previous method [17] and our method needed an average of 126 gates to implement the same circuits. As seen in the figure the proposed approach needs fewer gates than the previous approach in every range. The improvement in the gate count is greater for larger circuits.

The circuits generated by the method are on average 1% worse than the circuits generated by the method in [17]. The focus of this method is to reduce the gate count and in this regard the proposed method does better than the previous method for almost all circuits.

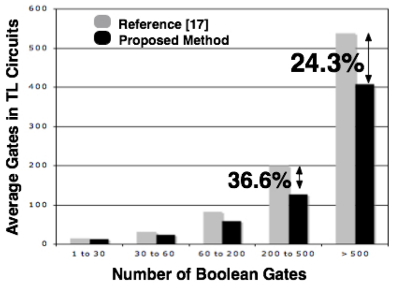


Fig. 11. Comparison of results

## VII. CONCLUSION

In this paper a novel theory for decomposition of Boolean functions into constituent threshold functions is proposed. New properties of threshold functions are introduced and proved. Using these properties we develop a decomposition procedure to identify and assign weights and threshold to a threshold function. A new algebraic factorization for factoring Boolean functions which is pivotal in making the proposed procedure work is presented. A new synthesis procedure is built on top of the decomposition theory. This procedure has sound theoretical basis as opposed to other heuristic node merging algorithms. When compared to the most recent synthesis procedure the proposed method generates threshold circuits that have an average of 27% fewer gates with comparable circuit depth.

## APPENDIX

*Lemma 1:* If $F = xA + B$ in a MML factor tree, and $A$ and $B$ are known to be threshold, with a weight-threshold assignment $[W; T_A]$ and $[W; T_B]$ (identical input weights) respectively and $T_B > T_A$, then $F$ is also a threshold function with $w_x = T_B - T_A$ and $F \equiv [W \cup \{T_B - T_A\}; T_B]$.

*Proof:* In [10] it is shown that $A = F_x$ and $B = F_{x'}$. Therefore, $F = x.A + x'.B$ (Shannon decomposition [12]). To prove the Lemma we need to show that for every one-point $O = \{o_1, o_2, \cdots, o_n\}$ of $F$, $\sum_{o_i \ \epsilon \ O} w_i o_i \geq T$, and for every zero-point $Z = \{z_1, z_2, \cdots, z_n\}$ of $F$, $\sum_{o_i \ \epsilon \ O} w_i o_i < T$.

**CASE 1** $x = 1$**:** $x = 1 \Rightarrow F = A$. $F(O) = 1$

$$\Rightarrow A(O \setminus \{o_x\}) = 1 \Rightarrow \sum_{o_i \ \epsilon \ (O \setminus \{o_x\})} w_i o_i \geq T_A$$

$$\Rightarrow \sum_{o_i \ \epsilon \ (O \setminus \{o_x\})} w_i o_i + (T_B - T_A)1 \geq T_A + T_B - T_A$$

$$\Rightarrow \sum_{o_i \ \epsilon \ O} w_i o_i \geq T_B.$$

$F = 0 \Rightarrow A = 0$. $F(Z) = 0 \Rightarrow A(Z \setminus \{z_x\}) = 0$

$$\Rightarrow \sum_{z_i \ \epsilon \ (Z \setminus \{z_x\})} w_i z_i < T_A \Rightarrow \sum_{z_i \ \epsilon \ (Z \setminus \{z_x\})} w_i z_i < T_B$$

$$(since \ T_A < T_B).$$

**CASE 2: When** $x = 0$**.** $x = 0 \Rightarrow F = B$. $F(O) = 1$

$$\Rightarrow B(O \setminus \{o_x\}) = 1 \Rightarrow \sum_{o_i \,\epsilon\, (O \setminus \{o_x\})} w_i o_i \geq T_B$$

$$\Rightarrow \sum_{o_i \,\epsilon\, (O \setminus \{o_x\})} w_i o_i + (T_B - T_A)0 \geq T_B$$

$$\Rightarrow \sum_{o_i \,\epsilon\, O} w_i o_i \geq T_B.$$

$F = 0 \Rightarrow B = 0$. $F(Z) = 0$

$$\Rightarrow B(Z \setminus \{z_x\}) = 0 \Rightarrow \sum_{z_i \,\epsilon\, (Z \setminus \{z_x\})} w_i z_i < T_B$$

$$\Rightarrow \sum_{z_i \,\epsilon\, (Z \setminus \{z_x\})} w_i z_i + (T_B - T_A)0 < T_B$$

$$\Rightarrow \sum_{z_i \,\epsilon\, (Z \setminus \{z_x\})} w_i z_i < T_B.$$

■

*Lemma 2:* In all weight-threshold assignments of a positive threshold function, the *don't care* variables have weight strictly less than all other non-don't care variables.

*Proof:* Let $d$ be a don't care variable of the function $F$. By definition the value of $F$ is the same when $d = 1$ and when $d = 0$. Let $a, b, c \,\epsilon\, Supp(F)$. To prove the Lemma we now need to show that $w_d < w_j, j \,\epsilon\, \{a, b, c\}$. Without loss of generality, suppose $w_d \geq w_a$.

Let $P_a$ be a prime implicant of $F$ that contains $a$. Since $F$ is a positive threshold function $w_j > 0, j \,\epsilon\, \{a, b, c\}$. Now $\sum_{i \,\epsilon\, P_a} w_i \geq T \Rightarrow \sum_{i \,\epsilon\, P_a; i \neq a} w_i + w_a \geq T$.

Since $w_d > w_a$; $\sum_{i \,\epsilon\, P_a; i \neq a} w_i + w_d \geq T$. $\therefore P_{a_{a \to d}}$ belongs to the *on-set* of $F$. Since $d$ is a don't care, changing $d$ from 1 to 0 will not change the output of $F$. Therefore by setting $d = 0 \Rightarrow P_a \setminus a$ is also in the *on-set* of $F$. $\therefore \sum_{i \,\epsilon\, P_a; i \neq a} w_i \geq T$

This implies that $P_a \setminus a$ is an implicant of $F$. If $P_a \setminus a$ is an implicant, then $P_a$ is not a prime implicant (by definition of prime implicant). This is a contradiction. Therefore $w_d \not\geq w_a$. Similarly $w_d \not\geq w_k, k \,\epsilon\, \{b, c\}$. Hence $w_d < w_j, j \,\epsilon\, \{a, b, c\}$. ■

Let $Ax + B$ is the *complete sum* of a positive threshold function $F$, where $x$ is the variable with the highest weight. Lemma 3, 4 and 5 discuss some properties of this representation of $F$.

*Lemma 3:* There exists a weight-threshold assignment of $A$ and $B$ such that both $A$ and $B$ have the same variable weights.

*Proof:* In [10] it is shown that if $x$ is the variable with the highest weight in the threshold function $F$ and $CS(F) = Ax + B$, then $A = F_x$ and $B = F_{x'}$.

Also if $F = [W; T]$, then $A = [W \setminus w_x; T - w_x]$ and $B = [W \setminus w_x; T]$ [10]. Notice that variable weights for both $A$ and $B$ are the same ($W$). Therefore there exists weight-threshold assignments in which both $A$ and $B$ have the same variable weights. ■

*Lemma 4:* Both $A$ and $B$ have the same $\succeq$-ordering of variables.

*Proof:* From the weight-threshold assignment of a threshold function, $\succeq$-ordering can be obtained using the following relationship between weights and $\succeq$-ordering [16]:
**(1)** If $w_x > w_y$, then $x \succeq y$. **(2)** If $w_x = w_y$, then $x \approx y$.

Since there exists a weight-threshold assignment for $A$ and $B$ such that they both have the same variable weights (Lemma 3), both $A$ and $B$ have the same variable $\succeq$-ordering. ■

*Lemma 5:* If $Supp(A) \setminus Supp(B) \neq \phi$ and $Supp(B) \setminus Supp(A) \neq \phi$, then $F$ is not a threshold function.

*Proof:* Suppose $F$ is a threshold function. Since $Supp(A) \setminus Supp(B) \neq \phi, \exists a \,\epsilon\, A \setminus B$. Similarly $\exists b \,\epsilon\, B \setminus A$. If $Supp(A) \cap Supp(B) \neq \phi, \exists c$ such that $c \,\epsilon\, Supp(A) \cap Supp(B)$.

It can be seen that $b$ is a don't care variable for function $A$ and $a$ is a don't care variable for function $B$.
From Lemma 2, for function $A, w_a > w_b$ For function $B, w_b > w_a$. From Lemma 3 we know that there is at least weight-threshold assignment such that $A$ and $B$ have the same variable weights.

But as shown before $w_a > w_b$ in $A$, and $w_a < w_b$ in $B$. This is clearly a contradiction of Lemma 3. Therefore $F$ is not a threshold function and the lemma is proved. ■

*Lemma 6:* In a complete sum of a threshold function, let $C_k$ represent the cubes with $k$ literals. For any two variables $x$ and $y$
**(1)** If $w_x > w_y$, then $x$ will appear more often than $y$ in $C_k$, with the smallest $k$ among the set of $C_k$'s. **(2)** If $w_x = w_y$, then $x$ and $y$ will appear equally often in each $C_k$.

*Proof:* Proof follows from Theorem 5.1.6 in [16] (*pg. 120*). ■

*Lemma 7:* If $f(X)$ is a threshold function and has a weight threshold assignment $[\{w_1, \cdots w_{a-1}, w_a, w_{a+1}, \cdots, w_n\}; T]$, then for $f(X; a \to a')$, $[\{w_1, \cdots w_{a-1}, -w_a, w_{a+1}, \cdots, w_n\}; T - w_a]$ is a feasible weight threshold assignment.

*Proof:* For proof and a detailed discussion refer [16] (*pg 58*). ■

## References

[1] International Technology Roadmap for Semiconductors. 2003.
[2] M. Avedillo and J. Quintana. A Threshold Logic Synthesis Tool for RTD Circuits. In *Euromicro Sym. on Dig. Syst. Design*, 2004.
[3] P. Avouris, J. Appenzeller, R. Martel, and S. J. Wind. Carbon nanotube electronics. *Proceedings of the IEEE*, 91(11):1772–1784, 2003.
[4] V. Beiu, J. M. Quintana, and M. J. Avedillo. VLSI implementations of threshold logic-a comprehensive survey. In *IEEE Transactions on Neural Networks*, volume 14, 2003.
[5] E. P. Blair and C. S. Lent. Quantum-dot cellular automata: an architecture for molecular computing. In *Proc. of SISPAD 2003*.
[6] P. Celinski, S. D. Cotofana, J. F. López, S. F. Al-Sarawi, and D. Abbott. State of the art in CMOS threshold logic VLSI gate implementations and systems. In *Proceedings of the SPIE*, 2003.
[7] M. Dertouzos. *Threshold Logic:A Synthesis Approach.* MIT Press, 1965.
[8] Ellen M. Sentovich et al. SIS: A System for Sequential Circuit Synthesis. Technical report, Department of EECS, UC Berkeley, CA, 1992.
[9] T. Gowda, S. Leshner, S. Vrudhula, and G. Konjevod. Synthesis of threshold logic using tree matching. In *Proc. of ECCTD*, 2007.
[10] T. Gowda, S. Vrudhula, and G. Konjevod. Combinational equivalence checking for threshold circuits. In *Proc. of the ACM GLSVLSI*, 2007.
[11] T. Gowda, S. Vrudhula, and G. Konjevod. A non-ilp based threshold logic synthesis methodology. In *Proc. of the IWLS*, 2007.
[12] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Boston: Kluwer Academic Publishers, 1996.
[13] Z. Kohavi. *Switching and Finite Automata Theory.* New York: McGraw-Hill Book Company, 1970.
[14] K. K. Likharev. Single-electron devices and their applications. *Proceedings of the IEEE*, 87(4):606–632, 1999.
[15] P. Mazumder, S. Kulkarni, M. Bhattacharya, J. P. Sun, and G. I. Haddad. Digital circuit applications of resonant tunneling devices. *Proceedings of the IEEE*, 86(4):664–686, 1998.
[16] S. Muroga. *Threshold Logic and Its Applications.* New York: WILEY-INTERSCIENCE, 1971.
[17] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha. Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies. In *IEEE Transactions on CAD*, 2005.