Synthesis of Threshold Logic Circuits Using Tree Matching

Tejaswi Gowda, Samuel Leshner, Sarma Vrudhula and Goran Konjevod
School of Computing and Informatics, Arizona State University, Tempe, AZ 85281
{tejaswi, samuel.leshner, vrudhula, goran}@asu.edu

Abstract— **Threshold logic has been known to be an alternative to Boolean logic for over four decades now. However, due to the lack of efficient circuit implementations, threshold logic did not gain popularity until recently. This change is motivated by new and efficient alternative CMOS implementations for threshold logic and futuristic nano devices like RTDs and SETs which possess inherent threshold properties. This paper motivates the need for threshold logic, and justifies it as an alternative design technique in the post-CMOS era. We present a novel synthesis algorithm for threshold circuits based on tree matching. In comparison with the previous state of the art methods the proposed method demonstrates an improvement of 25% in the number of gates required (max improvement is 50%) and comparable circuit depth.**

I. Introduction

Through continuous scaling for over 35 years, the dimensions of MOSFETs have reached atomic scales. After 10-15 years, the SIA roadmap [1] predicts that further scaling will not be sustainable, and expects a transition from CMOS to presently nascent nano technologies such as resonant tunneling diodes (RTDs)[12], carbon nanotube FETs (CNFETs) [4], and single electron transistors (SETs) [10].

Innovations in manufacturing technology are certainly crucial, but are not sufficient. The development of a new *design technology* which includes new circuits architectures, design methodologies, and CAD tools are part of the essential infrastructure needed to prepare for the post-CMOS era. An important and distinctive characteristic of these post-CMOS nano technologies is that they make it possible to efficiently and naturally implement threshold logic (TL). Moreover, TL can not only be be implemented efficiently in today's CMOS technology, but can also help mitigate the challenges posed by scaling, providing a means for seamless adoption of the new technology.

The basic concepts underlying TL are not new. Much of the earlier work on TL dates back to the 1960s [11], [6], [17], [9], which focused primarily on exploring the theoretical aspects with little attention being paid to the synthesis and optimization of large TL networks. Recently, however, there has been renewed interest in TL, and numerous methods have been proposed for the synthesis and verification of threshold circuits [14], [3], [13], [15], [16].

In this work, we present a new method for synthesizing a multi-level threshold network. We provide a clear justification for the statements made above on the various advantages of TL logic. Having justified the importance of TL logic, we proceed with description of the algorithm for synthesis of threshold networks, given a *factored form* (represented as a factor tree) of the function. The method partitions the factor tree into sub-trees such that each sub-tree represents a threshold function. The method makes use of a built in *library* of threshold gates for the determination of threshold functions. Using such a library, the algorithm uses the principles of dynamic programming with tree matching to generate a threshold network with the least number of threshold elements.

A. Main Contributions

The most important contribution of this work is that it makes use of a factored form of a logic function to generate the threshold circuit. To the best of our knowledge, we believe that this is this first method for synthesizing multi-level threshold networks starting from a functional representation. Most of the improvements generated can be attributed to this feature. This is in contrast to the most recently published method [14] for synthesis of threshold networks that uses a Boolean circuit as an input. In addition, the method to be described eliminates the need for using integer linear programming (ILP) [11] to determine whether a given function is threshold. We believe that this is an important step towards development of TL design automation methods that are independent of Boolean logic optimization.

II. Background and Motivation

Definition 1: A *threshold logic* (TL) gate has n binary inputs $x_1, \ldots, x_n$, and a single binary output y. Its internal parameters are a *threshold* T and weights $w_1, \ldots, w_n$, where weight w_i is associated with input x_i. The values of the threshold T and the weights w_i ($i = 1, \ldots, n$) may be any finite real numbers [6], [9], [11]. The input output relation of a threshold gate is defined as:

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

The *weighted sum* in Equation (1) denotes arithmetic summation. **Note:** A threshold function is one that can be realized by a single TL gate. It is completely characterized by the set of weights $W = (w_1, \ldots, w_n)$ and the threshold T, and hence is denoted by $[W; T]$.

Differential current-mode logic (DCML) [5] has been studied as an alternative to static CMOS TL design. DCML is a dynamic logic style that utilizes a differential current comparator cell to compare two currents over a selected time interval and output which of the two is larger. Figure 1 shows the general structure of the DCML gate. Each gate is comprised of two physically symmetrical cross-coupled networks: the input network and the threshold network. Each network consists of some number of parallel devices which are used to define the input weights and the threshold value of the function. Each gate is operated in two stages: the reset stage and the evaluation stage. While the gate is in the reset stage, both the output and negated output are pulled low. When the gate transitions to the evaluation stage, either the output or negated output is pulled high, depending on result of the threshold computation. Once the evaluation has completed, the outputs are effectively latched until the next reset stage.

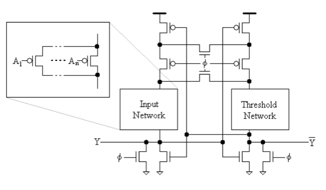

Fig. 1. DCML TL Gate Schematic

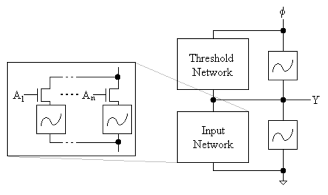

Fig. 2. NDR TL Gate Schematic

Area advantages: The transistor stack size of DCML is constant, thus increasing the size of devices to accommodate larger fan-in is not required. Additionally, threshold functions in DCML are defined merely by sizing the widths of the devices in the input and threshold networks. This approach may require far fewer devices than a static CMOS implementation (especially as the fan-in increases). Figure 3 compares the area of a DCML and static CMOS gate for fan-in up to 6 inputs. As indicated, the constant area overhead of DCML is quickly dwarfed by the rising area cost of CMOS as the fan-in increases.

Energy and delay advantages: As the size and number of devices in the gate increases, the energy required by the gate will naturally increase as well. Since DCML consumes no stand-by current, only the energy consumed per computation is considered. In CMOS implementations, delay increases as the transistor stack size increases, and decreases as the size of the devices increases. In DCML, the transistor stack size is constant. Figure 4 shows that the energy-delay product for the n-input NOR for static CMOS grows very quickly, while the energy-delay product for DCML grows very slowly.

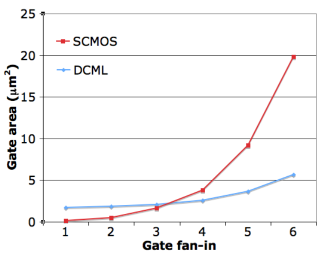

Fig. 3. Worst case gate area

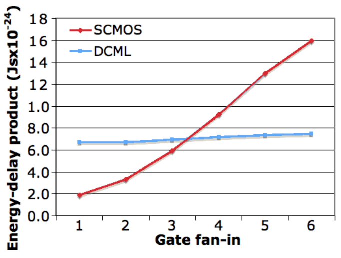

Fig. 4. EDP for NOR function

Transition to RTD based design: The greatest advantage of the DCML style is the ease with which one can transition from it to the logic style implemented by future nanoscale devices utilizing properties of negative differential resistance. Negative differential resistance, or NDR, is a characteristic of many nanoscale devices identified by a non-monotonic current flow as a function of differential voltage. This property enables individual devices to act as implicit memory elements. The monostable-bistable logic element, or MOBILE [12], has been studied extensively by researchers investigating resonant tunneling diodes and other NDR nanoscale devices. As Figure 2 shows, monostable-bistable NDR logic elements have the exact same general components as DCML. They possess an input network of parallel devices, a threshold network of parallel devices, a control signal which switches the logic element between a reset and evaluation stage, and the same self-latching output property. Since the input weights and threshold value are determined by parallel devices just as they are in DCML, any DCML gate may be simply replaced by an NDR gate.

III. Previous Work

Until recently, there has been no significant effort to develop efficient algorithms for synthesizing multi-level TL networks. Much of the earlier work related to TL synthesis [11], [9] focused on determining whether or not a given Boolean function is threshold, i.e. can be implemented using a single TL gate. The direct approach is to derive a set of linear inequalities or constraints from the truth table, remove the redundant constraints and see if the resulting set is consistent. If so, then linear programming is used to determine the *minimal weight* assignment. Techniques for synthesizing two-level networks using K-maps [9] have also been known for a long time but are not practical for circuits of present-day complexity. An algorithm for synthesis of two-level TL networks with weights restricted to be ± 1 is described in [2].

The problem of multi-level TL network synthesis has been investigated only recently [13], [14]. These techniques start with a Boolean network, synthesized using existing logic synthesis tools, and heuristically merge gates to form threshold gates. Whether or not gates can be merged to form TL gates is determined by repeatedly solving an ILP problem. The quality of the results are very sensitive to the input Boolean network. While these works address multi-level TL network synthesis, there is little that can be said about the optimality of the results.

The method proposed in [3] designs a generic multilayer forward feed threshold circuit by transforming the synthesis problem into a satisfiability problem. This method [3] does not limit the fan-in of gates. Limiting the fan-in is important, as fan-in cannot be arbitrarily large in a physical implementation. We limit our fan-in to six and compare our work with that of [14].

IV. Problem Formulation

A. Definitions and Notations

Maximally Factored Factor Form: A factored form is maximally factored [7] if **1)** for every sum of products, there are no syntactically equivalent factors in the product, and **2)** for every product of sums, there are no two syntactically equivalent factors in the sums.

Best Literal Factorization: This is a type of generic factoring algorithm [7] in which the algebraic divisor chosen is the literal that occurs in the greatest number of cubes. *e.g:* $a(b+c)+d$ is the best-literal factorization of the SOP $ab+ac+d$.

The Factor Tree: The factor tree is a binary expression tree which represents the factored form of a Boolean function. The leaf nodes of the factor tree correspond to the literals in the factored form, and the non-leaf nodes correspond to the conjunction and disjunction operators in the factored form. For example the factor tree for the function $(a+b')(c'+d)$ is shown in Figure 5.

Threshold Gate Library: The threshold gate library is a collection of all unique factor trees of threshold functions. For example, a threshold gate library with a fan-in restriction of three would consist of the following factor trees: ab, $a+b$, $a+bc$, $a(b+c)$, abc, $a+b+c$ and $a(b+c)+bc$. All functions are maximally factored. Here a, b and c are generic place holders for three unique literals. Each literal can occur in positive or negative form.

V. Tree matching using dynamic programming

The algorithm takes as input a maximally factored form of the function. The objective is to generate a circuit that implements the given function using the minimum number of threshold gates. We have at our disposal a library of all threshold gates of n or fewer inputs, for a specified n. We attempt to divide the entire factor tree into a set of disjoint sub-trees, such that each sub-tree corresponds to a tree present in the library. We can replace each sub-tree by the threshold gate it represents, thus obtaining our required circuit.

Example: Consider a *library* that has all 4 input threshold gates. Figure 6 shows a valid tree partitioning for the factor tree of $H=(ab+cd)e'$. Once the circuit structure is determined, we can trivially assign the predetermined weight and threshold to each element (which is determined when the library is built, and is done only once). Thus the issue of weight-threshold determination is eliminated in this approach. Assigning weights to negative literals is done as follows: if $f(X)$ realized as a threshold gate has the structure $[W;T]$, $f(X, x_p \rightarrow x_p')$ is realized with $[w_1, w_2, \cdots, -w_p, \cdots, w_n; T-w_n]$ (see Theorem 3.2.2 page 58 in [11]).

We first describe how the algorithm works for a single output function. When any functional representation of the function is given, the maximally factored form of the function is obtained. A factor tree is constructed for this factored form (using $getFactorTree$). The method is unlike the state-of-art TL synthesis method [14], which takes an optimized Boolean network as input.

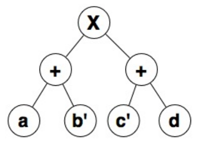

Fig. 5. Factor Tree

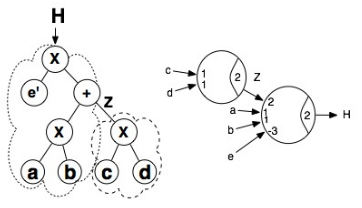

Fig. 6. Partitioned tree and circuit

A library, which has factor trees of all threshold functions with n or fewer inputs is also provided as input. Using the library the factor tree of the function is partitioned into sub-trees, such that each sub-tree corresponds to a factor tree in the library. Since all threshold functions are known beforehand, pre-determined weights are assigned to each gate. The factored tree is a global data structure.

Algorithm 2 (GETCOST) is the dynamic programming [8] formulation of the algorithm. Dynamic programming makes the procedure efficient by *memoizing* optimal solutions and eliminating redundant computations [8]. COSTMATRIX and SOLUTION are global data structures that store the intermediate results. Each of these arrays has one entry for each non-leaf node in the factor tree. Each entry in COSTMATRIX stores the cost (the number of gates) for implementing the function rooted at that node. SOLUTION stores the threshold gate that is to be used at that node in the least gate circuit. SOLUTION is necessary to recover the best solution, once the execution of the algorithm (GETCOST) is complete. The initial invocation of GETCOST takes in the root node of the factor tree as the argument.

Algorithm 1: Tree partitioning algorithm for TL Synthesis
Input: Maximally factored form (MFF), TL gate library.
Output: A threshold circuit implementing the function.
** *CostMatrix stores the min. gate count for each node.* **
** *Solution stores best solution for each node.* **
** *Initialize the elements in* COSTMATRIX *to* ∞ **

```
TREEPARTITION(node)
(1)     root = GETFACTORTREE(MFF)
(2)     GETCOST(root)
(3)     GENERATESOLUTION(Solution)
```

Algorithm 2: Least Gate TL Circuit Algorithm
Input: A node in the factor tree; TL gate library – Library.
Output: – *N/A* –

```
GETCOST(node)
(1)     if node is leaf
(2)         return 0
(3)     else
(4)         foreach Gate in Library
(5)             if Gate fits at node
(6)                 ** L = the list of nodes in factor tree that corresponds
                    to the leaves of GATE.**
(7)                 Cost = 1.
(8)                 foreach inpNode in L
(9)                     if CostMatrix[inpNode] $< \infty$
(10)                        Cost += CostMatrix[inpNode].
(11)                    else
(12)                        Cost += GETCOST(inpNode).
(13)                if Cost $<$ CostMatrix[node]
(14)                    CostMatrix[node] = Cost.
(15)                    Solution[node] = Gate
(16)        return CostMatrix(node)
```

Algorithm 3: Generating the solution TL circuit from the SOLUTION array
Input: A node in the factor tree – node
Output: – *N/A* –

```
GENERATESOLUTION(node)
(1)     Gate = Solution[node]
(2)     Generate the weight and threshold for the corresponding function
        in the factor tree, matching GATE
(3)     ** L = the list of nodes in the factor tree that correspond to the
        leaves of GATE.**
(4)     foreach inpNode in L
(5)         if inpNode in Solution
(6)             GENERATESOLUTION(inpNode)
```

The algorithm GETCOST calculates the minimal number of gates required to implement the given function. The algorithm also enumerates the actual solution. The solution is reconstructed from the SOLUTION data structure by the GENERATESOLUTION function.

The advantage of this dynamic programming formulation is that the best circuit implementation for each node is calculated only once. Since the cost and solution are stored, they can simply be read from their stored values rather than recomputed when they are required later, significantly improving the complexity of the algorithm.

A. Optimality of the Algorithm

Optimality of the algorithm is ensured by choosing the best circuit implementation for each node in the factor tree (lines 13 - 15 in Algorithm 2). Since SOLUTION stores the gate-minimal implementation for each node, after the algorithm completes execution the gate optimal solution for the root node is present in SOLUTION.

The reduction in the number of gates for a threshold circuit is more complex than for a Boolean circuit. After obtaining the minimal literal factored form, obtaining the minimal gate implementation is a non-trivial problem. In this regard, the proposed algorithm guarantees a gate minimal implementation once we decide on the factored form. The proposed algorithm gives the optimal gate-efficient implementation for a single output function. For a multi-output function, no such claim can be made, as the quality of the result depends on the initial logic optimization and extraction.

B. Complexity of the Algorithm

The time spent on a single call to GETCOST is $O(1)$ (the size of the library is finite and fixed, thus it is constant), excluding the time spent in the recursive calls it generates. So the running time is bounded by a constant times the number of calls issued to GETCOST. Since the algorithm gives no explicit upper bound on the number of calls, we try to find a bound by looking at a good measure of *progress* [8].

The most useful measure of progress is the number of entries in COSTMATRIX that are not *infinity*. Initially the number is zero. Each time the procedure is invoked by recursion, a new entry is filled. Since COSTMATRIX has only $O(n)$ entries, n being the number of nodes in the input factor tree, there can be at most $O(n)$ calls to GETCOST. Therefore the running time of GETCOST(node) is $O(n)$. GENERATESOLUTION and GETFACTORTREE (lines 1 and 3 in Algorithm 1) both have $O(n)$ complexity. Hence the TREEPARTITION algorithm has $O(n)$ complexity (*linear time complexity*).

C. Synthesis Flow For Multi-Output Functions

We now describe a synthesis procedure for multi-output functions that internally makes use of the tree partitioning algorithm. The procedure starts with a functional representation of the circuit. Next, logic optimization and extraction of common sub-functions from the circuit is performed. This generates a circuit graph in which each node is a logic function. The maximally factored form of each node is then generated. After generating the factor tree, the tree partitioning algorithm is used to generate the threshold network of each node. These individual circuits are combined, using the original circuit graph, to get the final TL circuit for the entire multi-output function.

VI. Experimental Results

The proposed algorithm was implemented in Python and was run on an Apple iBook with a G4 processor and 1 GB RAM. To generate the library we use the list of threshold functions provided in [11]. The list has all 2 to 5 input threshold networks. We generated a subset of six input threshold gates by using the five input threshold functions. If $f(x_1, ..., x_n)$ is a threshold function, then $y + f(x_1, ..., x_n)$ and $z.f(x_1, ..., x_n)$ are also threshold functions [11]. Next, we obtained the factored forms and generated factor trees for these functions. The collection of these factor trees constitutes the library, which is an input to our procedure.

TABLE I
Comparison with Previous Work

Bench-	Boolean Circuit		Method in [14]		Our Method	
mark	Gates	Levels	Gates	Levels	Gates	Levels
b1	10	4	8	3	6	4
decod	24	3	24	3	18	2
parity	45	9	45	9	30	8
f51m	101	8	82	8	44	4
9symml	141	10	110	9	85	10
cm162a	39	7	26	8	18	5
cm163a	40	6	25	6	15	4
tcon	32	3	32	3	16	2
cordic	61	9	49	7	30	6
ttt2	127	7	100	6	89	7
pcler8	50	7	47	7	34	9
frg1	97	12	59	9	40	7
my_adder	160	34	96	18	65	19
term1	278	11	226	10	118	9
cht	119	5	82	5	73	3
apex7	171	10	118	9	106	9
x1	293	8	203	7	104	7
example2	226	9	182	8	146	8
x4	264	7	189	8	176	6
apex6	543	12	396	12	326	10

Circuits in the MCNC benchmark suite were synthesized using the proposed method. A subset of the results are reported in Table I. Gate count and depth are non-technology specific metrics for area and delay. The table lists the results for the benchmark circuits when implemented as Boolean circuits and as TL circuits by the method described in [14]. Compared to the method in [14], our method generates circuits with comparable depth and 25% fewer gates on average (50% at best).

Histograms comparing results are shown in Figure 7. The x-axis of the first histogram represents the number of gates in the Boolean circuit. The y-axis plots the average number of gates in the threshold circuits generated by the two methods for the same benchmarks. 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 second histogram is similar but plots the average level of threshold circuits generated by the two approaches. Our method generates circuits with greater levels in some ranges and is comparable in other ranges. The objective of this method is to reduce the gate count and in this regard the proposed method does better than the previous method for nearly all circuits.

The comparison has only been done with the results reported in [14]. Comparison with the method in [3] is not meaningful because it does not impose a fan-in limit on gates. Both [14] and our method limit the fan-in of gates to six. The average runtime for the method is around 5 seconds. A runtime comparison with [14] is also not

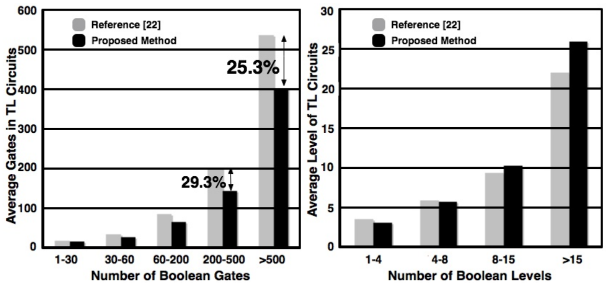

Fig. 7. Comparison of results

meaningful since their method uses an optimized Boolean network as input whereas the method presented herein starts with the basic logic representation. Runtimes reported in [3] are of the order of minutes (even for small circuits with three inputs).

The algorithm presented in this paper demonstrates a large improvement over Boolean logic implementations. It also demonstrates significant improvement over the previous TL synthesis method. The fan-in restriction we assume (six) is reasonable in light of device considerations. In the few circuits where we do worse, results can be improved by better logic extraction for threshold logic. The factor tree method may be extended to develop new TL extraction methods.

VII. Conclusion

A novel threshold logic synthesis method is proposed. The main contribution of this work is that it is the first step towards the development of an independent theory for TL CAD. Unlike other methods, this approach starts with a functional representation and not a Boolean circuit. It also eliminates the use of the ILP formulation by using pre-determined weights. Currently, the focus of this method is on reducing the number of gates. In the future, this method can be extended to design level optimal circuits and synthesis methodologies with area-delay trade-offs.

References

[1] International Technology Roadmap for Semiconductors. 2003.
[2] A. L. Oliveira et al. LSAT-An Algorithm for the Synthesis of Two Level Threshold Gate Networks. In *Proc. of ICCAD*, 1991.
[3] M. Avedillo and J. Quintana. A Threshold Logic Synthesis Tool for RTD Circuits. In *Euromicro Symposium on Digital System Design*, 2004.
[4] P. Avouris, J. Appenzeller, R. Martel, and S. J. Wind. Carbon nanotube electronics. *Proceedings of the IEEE*, 91(11):1772–1784, 2003.
[5] S. Bobba and I. Hajj. Current-Mode Threshold Logic Gates. In *Proceedings of the IEEE ICCD*, pages 235–240, 2000.
[6] M. Dertouzos. *Threshold Logic:A Synthesis Approach*. MIT Press, 1965.
[7] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Boston: Kluwer Academic Publishers, 1996.
[8] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005.
[9] Z. Kohavi. *Switching and Finite Automata Theory*. New York: McGraw-Hill Book Company, 1970.
[10] K. K. Likharev. Single-electron devices and their applications. *Proceedings of the IEEE*, 87(4):606–632, 1999.
[11] S. Muroga. *Threshold Logic and Its Applications*. New York: WILEY-INTERSCIENCE, 1971.
[12] P. Mazumder et al. Digital circuit applications of resonant tunneling devices. *Proceedings of the IEEE*, 86(4):664–686, 1998.
[13] Rui Zhang et al. Synthesis of Majority and Minority Networks and Its Applications to QCA, TPL and SET Based Nanotechnologies. In *VLSID'05*.
[14] Rui Zhang et al. Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies. In *IEEE Transactions on CAD*, 2005.
[15] T. Gowda et al. A Non-ILP Based Threshold Logic Synthesis Methodology. In *Proc. of the 16th IWLS*, pages 222–229, 2007.
[16] T. Gowda et al. Combinational equivalence checking for threshold logic circuits. In *Proc. of the 17th GLSVLSI*, pages 102–107, 2007.
[17] R. O. Winder. *Threshold Logic*. PhD thesis, Princeton University, 1965.