Combinational Equivalence Checking for Threshold Logic Circuits

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

ABSTRACT

Threshold logic is gaining prominence as an alternative to Boolean logic. The main reason for this trend is the availability of devices that implement these circuits efficiently (current mode, differential mode circuits), as well as the promise they hold for the future nano devices (RTDs, SETs, QCAs and other nano devices). This has generated renewed interest in the design automation community to design efficient CAD tools for threshold logic. Recently a lot of work has been done to synthesize threshold logic circuits. So far there has been no efficient method to verify the synthesized circuits. In this work we address the problem of combinational equivalence checking for threshold circuits. We propose a new algorithm, to obtain compact functional representation of threshold elements. We give the proof of correctness, and analyze its runtime complexity. We use this polynomial time algorithm to develop a new methodology to verify threshold circuits. We report the result of our experiments, comparing the proposed methodology to the naive approach. We get up to $189X$ improvement in the run time ($23X$ on average), and could verify circuits that the naive approach could not.

Categories and Subject Descriptors: B.6 [Hardware] : Logic Design ; B.6.3 [Logic Design]: Design Aids – *Verification.*

General Terms: Algorithms, Design, Theory, Verification.

Keywords: EDA, Equivalence checking, Nano devices, Threshold logic.

1. INTRODUCTION

For more than four decades, digital circuits have been implemented by representing Boolean functions as a network of AND, OR and NOT logic gates, and an enormous amount of research and development has taken place in synthesis, optimization, testing and verification for such networks. It has also been known, for an equally long period, that there is an alternative approach in which the logic primitives are replaced by elements of a much larger class of functions known as *threshold* logic gates [7, 11, 15]. For instance, each of the Boolean functions $ab(c+d)+cd(a+b)$ and $a(b+c+d)+b(c+d)+cd$, can be realized by a *single threshold gate*. Thus a Boolean function, if realized as a network of threshold gates, can result in significantly fewer nodes and smaller network depth. Unfortunately, the lack of an efficient implementation of a threshold gate during the early history of digital systems led to the dominance of AND/OR networks, and consequently, the heavy investment in design tools. CMOS circuits implementing Boolean logic have higher *functional yield* in the presence of process variations, when compared to threshold circuits. This has also contributed to the popularity of Boolean logic.

The situation is changing in favor of threshold logic. First, recently there have a number of very efficient implementations of threshold logic gates in CMOS[4, 3, 2] that have achieved very high performance and significantly reduced area. Second, several new and promising nanoscale device technologies such as resonant tunneling diodes (RTD), quantum cellular automata (QCA), single electron transistors (SET) and others, are either fundamentally threshold type devices, or can be used to efficiently implement threshold logic[16]. Third, methods for network synthesis, optimization, testing and verification of threshold logic can now benefit from the knowledge gained over the past two decades through the development of the same for AND/OR networks. For these reasons there has been a renewed interest in developing algorithms for synthesis of threshold networks[1, 19, 18].

Synthesis is the process of transforming one representation of a function to another, usually to a more detailed, specification. Closely related to synthesis is *equivalence checking* (EC), which entails demonstrating the *equivalence* of two representations. The function of a circuit synthesized by software tools needs to be verified against the given functional specification. In addition, *engineering changes* that are introduced throughout the design process can introduce errors in the synthesized circuit. Demonstrating equivalence may also be required between two different representations of the same circuit generated at different phases in the design flow.

Equivalence checking of logic networks is a well developed and mature subject[13]. However, the same is not the case with threshold networks. In fact, it was only recently [1, 19, 18] that the problem of threshold network synthesis has started to become a subject of active research. While much more remains to be done on optimal synthesis, there has been no work that addresses the problem of verifying these synthesized threshold circuits. To the best of our knowledge, we are not aware of any non-trivial method to determine the logic function implemented by a threshold gate. Given the resurgence of interest in threshold networks, an efficient method for checking the equivalence of a threshold network and a logic network or its functional specification is of great value.

1.1 Main Contributions

We propose the first efficient procedure to determine the logic function of a threshold gate. The procedure is provably

GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.

correct and has polynomial total complexity. It generates a maximally factored form representation of the logic function, which is very compact. This results in the significant speed-up($1.25X$ to $16X$) when equivalence checking is done using BEDs[9]. Moreover the maximally factored form generated is that of a minimal SOP (which is the complete sum for a unate function[8]). This results in much smaller representation using BEDs.

2. BACKGROUND AND PREVIOUS WORK

DEFINITION 1. *A threshold element or gate has n binary inputs $x_1, x_2, \ldots, x_n$, and a single binary output y. Its internal parameters are a threshold T and weights $w_1, w_2, \ldots, w_n$, where weight w_i, is associated with input x_i. The values of the threshold T and the weights w_i $(i = 1, 2, \ldots, n)$ may be any real, finite, positive or negative number [7, 11, 15]. The input output relation of a threshold gate is defined as follows:*

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

The *weighted sum* in Equation (1) denotes arithmetic summation. **Note:** A threshold function is completely characterized by the set of weights $W = (w_1, \ldots, w_n)$ and the threshold T. Hence a threshold function is denoted by the pair $[W; T]$.

Example: Figure 1 shows a three input threshold gate. The weights associated with inputs a, b, c are $-2, 1$ and 1 respectively and the threshold $T = 1$. The gate's output will be 1 for all input combinations for which $-2a+b+c \geq 1$, and 0 otherwise. If $a = 1$, then no combination of b and c will satisfy the inequality. If $a = 0$ then either $b = 1$ or $c = 1$ will satisfy the inequality. Therefore the logic function realized by this threshold element is $y = a'(b + c)$.

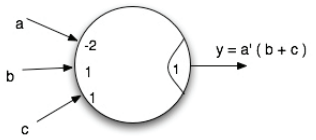

Figure 1: A threshold element implementing the function $y = a'(b + c)$

The algorithm to determine the logic function realized by a threshold gate does so by generating a maximally factored form. Below is a list of some basic definitions related to factored forms. For further details, the reader is referred to Hachtel et al. [8].

Maximally Factored Form: A factored form is maximally factored, if

1. for every sum of products, there are no two syntactically equivalent factors in the product,
2. for every product of sums, there are no two syntactically equivalent factors in the sums.

Complete Sum: An SOP formula is a complete sum (a sum of all prime implicants and only prime implicants) *iff*:

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 $a + c$.

Exact Factored Form: An exact factored form of an SOP is a factored form which when expanded by repeated algebraic multiplication only (without absorbing terms), will result in the original SOP. For example, consider the SOP form $F = ab + bc + ca$. The factored form $a(b + c + bc) + bc$ is not an exact factored form even though $F \equiv a(b + c + bc) + bc$. The factored form $a(b + c) + bc$ is an exact factored form of F.

Iterated Consensus: Iterated consensus is a method, based on the *consensus theorem* [8], and generates the complete sum of a function using any SOP. This method adds to the SOP, all the consensus terms of all pairs of cubes in the SOP. It then removes the terms that are present in other terms. This procedure is repeated until no further consensus is possible. *E.g.:* Consider the SOP of F, $x_1x_2 + x_2'x_3 + x_2x_3x_4$. *Iterated Consensus(F)* $= x_1x_2 + x_2'x_3 + x_1x_3 + x_3x_4$, the $CS(F)$.

The problem of demonstrating equivalence of two Boolean functions f and g, or their combinational circuit representations, has been extensively studied [13, 12, 14]. The "straightforward approach" is to construct an OBDD [6] of $f \equiv g$, which reduces to **1** if they are equivalent. The drawback of this approach is that construction of the OBDDs of f or g may not be efficient because their size may be exponential in the number of variables, regardless of the variable ordering (e.g. multiplier).

An alternative approach is to use to intermediate representations such as the AND/INVERTER graph [12] or Boolean Expression Diagram (BED) [9], that allow us to exploit the structural similarities that exist between the functions being compared. Since our implementation is based on BEDs [9], we limit the discussion only to them.

2.1 Boolean Expression Diagram

BED is a data structure obtained by extending the OBDD representation with operator vertices. A BED is similar to a logic graph representation of a Boolean circuit, with each gate replaced by an equivalent operator node and each input replaced by the corresponding variable node. All variable nodes are connected to the two terminal nodes (**0** and **1**). Figure 2 gives the example of a BED for the miter of the two circuits that are being compared.

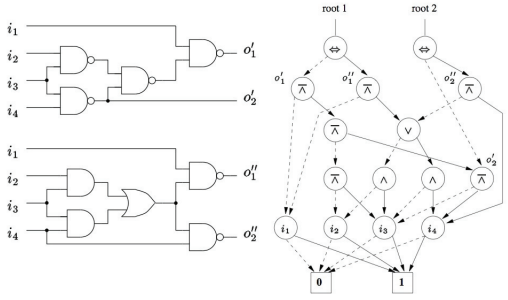

Figure 2: Boolean expression diagram example [9]

A BED representation is not canonical but is polynomial in size of the original circuit. Equivalence checking of two functions f and g is done by constructing the BED of their

miter [5] (Figure 3). Hulgaard et al.[9] present efficient transformations to reduce the size of the miter. The (significantly) reduced miter, can then be efficiently transformed into a OBDD [9], resulting in the equivalence check. The advantage of using BEDs is two fold. First, they provide for efficient hashing to simplify and speedup identification of structurally isomorphic parts of the two circuits. Second, it avoids creating the individual OBDDs for f and g, and constructs the OBDD of the reduced BED of the miter directly. This leads to a significant improvement in performance over the OBDD based approach. In fact, it often allows equivalence checking of circuits that have exponential size OBDDs.

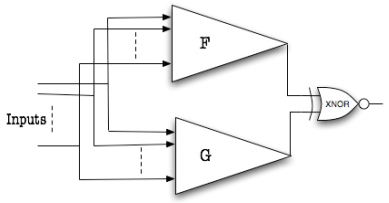

Figure 3: The *miter* of circuits F and G.

3. PROBLEM STATEMENT AND APPROACH

The problem addressed in this paper is to determine the equivalence of two threshold networks f and g. At least one of f or g is given in the form of a threshold network. The other may be logic network, a threshold network, or a functional specification of the circuit. We assume f and g have the same set of inputs and outputs, i.e., the mapping between the inputs and outputs of the two circuits is known.

A key step in verifying equivalence of threshold networks is the determination of the logic function realized by a threshold gate. Once this is done, then we can construct a logic network with the threshold gate replaced by its logic function. Then we can proceed with the construction of the BED of the miter to determine equivalence of the two networks.

The naive way to determine the logic function of a threshold gate is to try all 2^n input combinations and determine the *on-set* of the function, and generate a SOP representation. One of the features of threshold gates is that they permit efficient realization (both in area and delay) of gates with large fan-in. Hence the naive approach will not be practical. For instance, consider an n-input majority function which can be implemented as a single threshold gate. For $n = 16$, using the naive approach, takes over six minutes to generate the logic function and about eight seconds to verify equivalence (see Section 5). For $n = 24$, the naive approach takes more than a day and does not complete execution. In contrast, the method to be described takes about nine minutes.

Since the subsequent step of equivalence checking relies on the use of BEDs, it is important to generate a compact logic network representation of a threshold gate, as this will reduce the size of the BED. Hence, instead of generating an SOP form of a threshold gate, it is most important to generate a *maximally* factored form. It would be best if we had the maximal factored form of the minimal SOP (which is the complete sum for a unate/threshold function[8, 11]). The algorithm described herein does exactly that – it generates a maximally factored form of a minimal SOP for a threshold gate directly, without explicitly enumerating all the minterms or generating the complete sum.

4. THE ALGORITHM TG2MFF

The algorithm to determine a maximally factored form of a threshold gate is referred to as TG2MFF. It takes an n-input threshold function $F = [W; T]$, where $W = (w_1, w_2, \ldots, w_n)$, and the support set is $X = (x_1, x_2, \ldots, x_n)$. Let $W \setminus w_k = (w_1, w_2, \ldots w_{k-1}, w_{k+1}, \ldots, w_n)$. TG2MFF recursively decomposes F using cofactors. Its pseudo code is given in Algorithm 1.

Algorithm 1: Pseudo code of TG2MFF
Input: $F = [W; T]$ is a threshold function
Output: A maximally factored form
TG2MFF(F)

```
        ** W = [w_1, ..., w_n], X = [x_1, ..., x_n] **
        ** T is the threshold **
(1)     if n = 1
(2)         if w_1 ≥ T and T ≤ 0
(3)             return 1;
(4)         if w_1 ≥ T and T > 0
(5)             return x_1;
(6)         if w_1 < T and T ≤ 0
(7)             return x_1';
(8)         if w_1 < T and T > 0
(9)             return 0;
(10)    else
(11)        if  Σ_{∀w_j<0} w_j ≥ T
(12)            return 1;
(13)        if  Σ_{∀w_j>0} w_j < T
(14)            return 0;
(15)        ** w_k is the largest absolute weight **;
(16)        F_1 = [W \ w_k, T − w_k];
(17)        F_2 = [W \ w_k, T];
(18)        if w_k > 0
(19)            return x_k· TG2MFF(F_1) + TG2MFF(F_2);
(20)        else
(21)            return TG2MFF(F_1) +x_k'· TG2MFF(F_2);
```

Statements 2 through 9 constitute the terminal cases and are easily verified. The other two terminal cases (statements 11 to 14) can be verified using the fact that all minterms are in the *on-set* of **1** and no minterm is in the *on-set* of **0**. Selecting a variable whose absolute weight is maximum is **necessary** in order to obtain a maximally factored form.

Example: Consider $F(a, b, c) \equiv [2, 1, -1; 2]$, with $w_a = 2, w_b = 1, w_c = -1$ and $T = 2$. Applying TG2MFF we get:

$$\begin{aligned} F &= [2, 1, -1; 2] = a \cdot [1, -1; 0] + [1, -1; 2] \\ &= a\{b[-1; -1] + [-1; 0]\} + 0 = a\{b(1) + c'\} \\ &= a(b + c') \end{aligned}$$

It can be seen that $[2, 1, -1; 2]$ is a feasible assignment for the function $a(b + c')$.

4.1 Proof of Correctness

As can be seen, TG2MFF is very simple. However, the proof of correctness, which is essential, is not obvious. We first state a useful property of the co-factors of a threshold function. The proof of this appears in [15].

LEMMA 1. *Let* $\mid w_k \mid \geq \mid w_i \mid, \forall i$. *Suppose that* F *is positive unate in* x_k. *Let* $CS(F) = Ax_k + B$. *Then* $A + B = A$.

PROOF. Refer to Theorem 5.1.7 (pg. 121) in [15], from which the proof follows. □

Lemma 1 is also true if F is negative unate in the variable with the maximum weight. The proof is similar.

LEMMA 2. *Let* $F \equiv [W; T]$. *Algorithm TG2MFF*(F) *generates an exact factored form of* $CS(F)$.

PROOF. We first show, by induction, that the factored form generated by TG2MFF evaluates to the Boolean function represented by $[W; T]$. It is trivial to verify that for the terminal cases ($n = 1$), TG2MFF produces a factored form that evaluates to same function as $[w, T]$.

Let w_k be the weight largest in magnitude, and assume $w_k > 0$. The proof for $w_k < 0$ is similar. From Equation 1 we see that setting $x_k = 1$ and $x_k = 0$ yields $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$, respectively. Assume that TG2MFF, when supplied with $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$, produces the factored forms for F_{x_k} and $F_{x_k'}$, which are the positive and negative cofactors of F. Examining the pseudo code, TG2MFF when supplied with $[W; T]$ produces the factored form $x_k F_{x_k} + F_{x_k'}$. We want to show that this evaluates to the function denoted by $[W; T]$.

Let F be the function that $[W; T]$ represents. By Shannon decomposition, $F = x_k F_{x_k} + x_k' F_{x_k'}$. Since F is positive unate in x_k, $CS(F) = Ax_k + B$. Computing the cofactors of F using $CS(F)$ results in $F_{x_k} = A + B$ and $F_{x_k'} = B$. By Lemma 1, $A + B = A$. Therefore, $F_{x_k} = A$. Hence $CS(F) = x_k F_{x_k} + F_{x_k'}$. We have shown that what TG2MFF computes, evaluates to $CS(F)$, which is a representation of F.

We now show that TG2MFF produces an exact factored form. Since $CS(F) = Ax_k + B$, A and B must each be complete sums. By induction, TG2MFF produces exact factored forms for $[W \setminus w_k; T - w_k]$ and $[W \setminus w_k; T]$. These, if multiplied out would be the complete sums A and B, respectively. Therefore, $x_k[W \setminus w_k; T - w_k] + [W \setminus w_k; T]$ is an exact factored form of $CS(F)$. □

THEOREM 1. *TG2MFF generates a maximally factored form of the complete sum of the given threshold function.*

PROOF. The factorization $F = Q \cdot D + R$, obtained by dividing F by D (to get quotient Q and remainder R. Q, D and R are repeatedly factored), will result in a maximally factored form, if the following two conditions hold [8]:

1. If Q is a single cube then no literal in Q occurs in any cubes of R, and
2. If Q has more than one cube, then there is no factor of Q that is also a factor of R.

We prove that, the two conditions sufficient for maximal factorization are satisfied by the TG2MFF algorithm. Let $F \equiv [W; T]$, and w_k be a largest magnitude weight. As before, we assume $w_k > 0$. The proof of $w_k < 0$ is the same.

By factoring out x_k in $CS(F)$ we get $CS(F) = Ax_k + B$. Note that $B \leq A$ since $A + B = A$ by Lemma 1.

Case 1: Suppose A is a single cube. Since $B \leq A$, $B = AC$, where $C = C_1 + C_2 + \cdots + C_n$. Therefore $F = A(x_i + C_1 + C_2 + \cdots + C_n)$. Since A is a single cube, it must have at least one literal, say y. $A = Qy$. Hence $F = Qy(x_k + C_1 + C_2 + \cdots + C_n)$.

A one-point of F is $Q = 1, y = 1, x_k = 0, C_i = 1, C_j = 0, i \neq j$, for some i, j. Therefore

$$\sum_{\ell \in Q} w_\ell + w_y + \sum_{\ell \in C_i} w_\ell \geq T$$

Since $w_k \geq w_y$,

$$\sum_{\ell \in Q_i} w_\ell + w_k + \sum_{\ell \in C_i} w_\ell \geq T$$

This implies that that $y = 0$ is in the onset of F, which is not possible. Therefore A cannot be a single cube. Hence the first condition required for a maximal factorization is satisfied.

Case 2: Now suppose A has at least two cubes and A and B have a common factor. Therefore, let $A = (X_1 + X_2 + \cdots + X_a)(Y_1 + Y_2 + \ldots Y_b)$ and $B = (X_1 + X_2 + \cdots + X_a)(Z_1 + Z_2 + \ldots Z_c)$.

Note because the factorization is *algebraic*, none of the X_i and Y_i have a common literal and none of the X_i and Z_i have no common literal. Rewriting F, we have

$$F = (X_1 + \ldots + X_a)[(Y_1 + \ldots + Y_b)x_k + (Z_1 + \ldots + Z_c)]$$

A one-point of F is $Z_i = 1, X_j = 1, Z_p = 0, X_q = 0, x_k = 0$, for some i, j, and $\forall p \neq i, \forall q \neq j$. Hence,

$$\sum_{\ell \in Z_i} w_\ell + \sum_{\ell \in X_j} w_\ell \geq T.$$

$$\sum_{\ell \in Z_i} w_\ell + w_{X_{j_1}} + \ldots + w_{X_{j_d}} + \cdots + w_{X_{j_r}} \geq T.$$

Since $w_k \geq w_\ell, \forall \ell \ \epsilon X_j$.

Replacing $w_{X_{j_d}}$, by w_k, we obtain

$$\sum_{\ell \epsilon Z_k} w_\ell + w_{X_{j_1}} + w_{X_{j_2}} + \ldots w_k + \cdots + w_{X_{j_r}} \geq T.$$

This implies that $X_j = 0, Z_i = 1, x_k = 1, Z_p = 0$, for some i, and $\forall j$, every $p \neq i$ belongs to the onset of F, which is not possible. Therefore A cannot have a factor that is a factor of B, when A has more than one cube. Hence TM2MFF produces a maximally factored form of the complete sum of F, using the feasible weight assignment of F. □

4.2 The Verification Procedure

The equivalence checking procedure starts with two threshold networks. The maximally factored form of each threshold element in the network is obtained by the algorithm TG2MFF. These factored forms are used to construct the BED for each output of the two functions. As mentioned earlier the correspondence between the outputs of the two functions is known. This information is used to construct the BED of the *miter* for each output pair. The ROBDD of the *miter* is then obtained by using the BED package[17]. The BED package has efficient algorithms to convert a BED into an equivalent ROBDD. The outputs are equivalent if the ROBDD of the *miter* is the constant **1**. If all outputs of the two circuits are verified to be equivalent then the entire circuit is equivalent. To verify two circuits when one of them is Boolean and the other is threshold, a similar approach is followed.

An Example: Consider the threshold circuit shown in Figure 4. Assume that a synthesis tool generated this circuit when it was given the following specification:

$f = d(ab' + ac' + b'c)$; $e = (a + b)(a' + b')$.

To verify the two circuits by the method proposed, we first get the factored form of each node in the threshold circuit. Using $TG2MFF$ algorithm, we get the following factored forms, for each node: $X2 = d(b' + c')$; $f = X2(c + a)$; $X1 = a'b'$; $X0 = X1 + ab$; $e = X0'$.

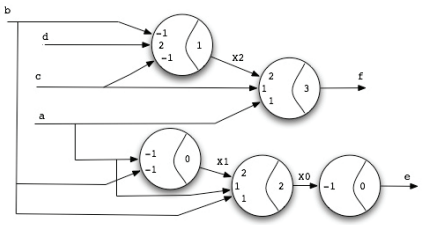

Figure 4: A generated threshold circuit

Using these factored forms and the circuit specification, BEDs of the miters are constructed. Since the circuits being compared here have two outputs we get two miters (*root 1* and *root 2* in Figure 5). The ROBDD of these two miters are constructed using the BED package. In our case the ROBDD of root 1 and root 2, turn out to be the constant **1**. Thus we can conclude that the threshold circuit synthesized is according to the specification. The verification of two threshold circuits is done in a similar way.

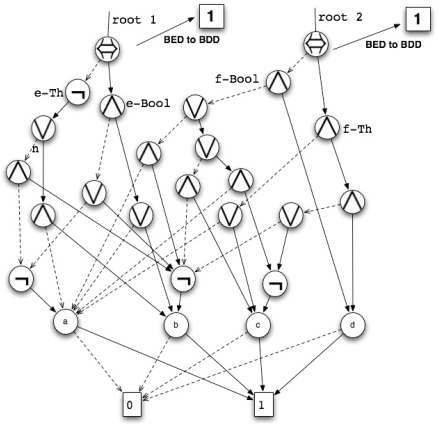

Figure 5: Example of threshold circuit verification

4.3 Complexity Analysis

TG2MFF generates a maximally factored from given a single threshold element. Hence its time complexity depends only on the size of the input and output, the size of the output being much larger than that of its input. Consequently, the time complexity is expressed in terms of the size of both inputs and outputs. This is typically done for algorithms, whose output size is much larger than the input size [10].

Let n and N be the number of literals in the input and the output respectively. When a terminal case (statements $2 - 9$ and $11 - 14$) is encountered TG2MFF halts the recursion and in the non-terminal case (statements $15 - 21$), it continues the recursion. At each stage (whether terminal or non-terminal) TG2MFF spends $O(n)$ time. This includes the checking for terminal cases and the time taken to invoke the next stage. At each non-terminal stage a new literal is added to the generated factored form. Thus the number of non-terminal stages is $O(N)$.

Each non-terminal stage can generate at most two terminal stages. Since the number of non-terminal stages is bounded by N, the number of terminal stages is also $O(N)$. Thus the number of invocations of the algorithm (sum of terminal and non-terminal stages) is $O(N)$. As said earlier since the algorithm spends $O(n)$ in each stage, the total complexity of TG2MFF is $O(nN)$. Hence the total complexity of TG2MFF is polynomial in the combined size of input and output. After the Boolean factor form is generated the BED generation can be done in linear time, since BED is just another representation of the factor form [9].

5. EXPERIMENTAL RESULTS

The few synthesis methods that have appeared in the literature recently [1, 18] generate circuits with high fan-in gates. However due to the unavailability of these tools and benchmarks for threshold logic, we generated our own benchmark circuits using the existing MCNC circuits. Since the bottleneck of the naive verification procedure (e.g. exhaustive enumeration of minterms) when applied to a threshold network is the fanin of gates and **not** the number of gates, we generated threshold networks with large fanin threshold elements. We used the directed acyclic graph representation of each MCNC benchmark circuit and replaced each node with a threshold element. This provided a complex threshold network. The weights and threshold of each threshold element were generated randomly.

Once the threshold networks were constructed, an equivalence check of each circuit with itself was done, to examine the running time of two procedures (TG2MFF vs exhaustive enumeration). After deriving the logic function, the BED tool was used to check the equivalence. The experiments were run on a Sun Fire V880 machine with 16GB RAM.

Table 1 lists the running time required for verifying the circuits by the proposed and naive method. There are two columns corresponding to each method. The first is the runtime to generate the function and the second is the time taken for the BED based verification. As seen from the Table, TG2MFF is more than an order of magnitude faster than the naive method. TG2MFF verified the 24 input majority gate in nine minutes, whereas the naive approach could not complete execution even after twenty four hours. TG2MFF takes much longer to generate the factor form of majority-24, when compared to the time taken to generate the factor form for majority-16, even though the input to the algorithm in the former case is only 1.5 times that of the latter case. This is because of the large differences in their factor forms (the output of TG2MFF).

TG2MFF also reduces the time required for the BED based verification. It runs **22X** faster on average and speeds up the BED based verification by **5X** on average (for circuits that could be verified by both methods). The first speed up is because of the polynomial total complexity of TG2MFF. The second speed up is due to the compact function representation produced by the algorithm. We note here that the factored form produced by TG2MFF is compact in two ways.

Table 1: Runtime Comparison

Benchmark Circuits	Inputs/ Outputs	Avg. Fanin	Max. Fanin	**A** : TG2MFF (sec)	**B** : BED (tg2mff) (sec)	**C** : Naive (sec)	**D** : BED (naive) (sec)	C / A	D / B
f51m-t	14 / 11	5	10	0.093	0.040	0.137	0.050	1.47	1.25
z4ml-t	19 / 9	6	10	0.099	0.040	0.181	0.060	1.83	1.5
cmb-t	19 / 15	3	6	0.116	0.040	0.218	0.080	1.88	2
cu-t	24 / 21	3	6	0.210	0.060	1.075	0.200	5.12	3.33
pcle-t	16 / 4	3	6	0.090	0.040	0.123	0.050	1.37	1.25
sct-t	47 / 36	3	6	0.126	0.040	0.239	0.070	1.90	1.75
majority-8	8 / 1	8	8	0.342	0.120	7.957	1.140	23.27	9.5
cht-t	9 / 1	4	6	0.459	0.130	22.149	1.760	48.25	13.54
cm152a-t	8 / 8	11	11	0.140	0.050	1.078	0.290	7.70	5.8
ttt2-t	7 / 4	3	9	0.110	0.040	0.417	0.140	3.79	3.5
x2-t	10 / 7	5	12	0.084	0.030	0.154	0.050	1.83	1.67
9symml-t	11 / 1	5	13	0.146	0.060	1.109	0.120	7.60	2
majority-16	16 / 1	16	16	1.975	0.470	374.168	7.790	189.45	16.57
majority-24	24 / 1	24	24	413.994	92.540	>1 day	–	–	–
Average Improvement								**22.73X**	**4.9X**

First because it is the factored form of the minimal SOP (the complete sum for a unate function). Secondly, because of the maximal factorization, the generated BED is compact. It can be observed that the time required for verification by the naive approach, depends on the fan-in of the individual gates and not necessarily on the number of gates. *Example:* Even though f51m-t has 8 gates it can be verified within a second, whereas majority-24, which has only one gate could not be verified in a day. This is because of the huge fan-in of the one gate in the majority-24 circuit.

6. CONCLUSION

In this paper we presented a new algorithm, TG2MFF, to generate a compact functional representation of a threshold element, and a proof of its correctness. TG2MFF has polynomial time complexity in the combined size of input and output. We also demonstrated the use of this algorithm to solve the problem of combination equivalence checking for threshold circuits. To the best of the our knowledge this is the first non-trivial threshold circuit verification methodology. The methodology can be further enhanced to improve performance by incorporating hashing in the recursion algorithm, to reduce redundant computations. The experiments we did on a number of generated threshold circuits show that the methodology gives up to $189X$ improvement in runtime as compared to the naive equivalence checking method. TG2MFF can be used to design better methods for decomposition, synthesis and verification of threshold circuits.

7. REFERENCES

[1] M. Avedillo and J. Quintana. A Threshold Logic Synthesis Tool for RTD Circuits. In *Euromicro Symposium on Digital System Design*, 2004.

[2] V. Beiu et al. VLSI implementations of threshold logic-a comprehensive survey. In *IEEE Transactions on Neural Networks*, volume 14, 2003.

[3] Y. Beiu et al. Differential Implementations of Threshold Logic Gates. In *Proceedings of the IEEE International Symposium on Signals, Circuits and Systems*, 2003.

[4] S. Bobba and I. Hajj. Current-Mode Threshold Logic Gates. In *Proc. of ICCD*, 2000.

[5] D. Brand. Verification of Large Synthesized Designs. In *Proc. ICCAD*, 1993.

[6] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume 35, 1986.

[7] M. Dertouzos. *Threshold Logic : A Synthesis Approach*. The MIT Press, 1965.

[8] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Boston: KAP, 1996.

[9] H. Hulgaard et al. Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams. In *IEEE Transactions on CAD*, July 1999.

[10] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005.

[11] Z. Kohavi. *Switching and Finite Automata Theory*. New York: McGraw-Hill Book Company, 1970.

[12] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *Proc. of DAC*, 1997.

[13] A. Kuehlmann and C. A. van Eijk. *Logic Synthesis and Verification*, chapter Combinational and Sequential Equivalence Checking. KAP, 2001.

[14] W. Kunz and D. Pradhan. Recursive Learning: A New Implication Technique for Efficient Solution to CAD Problems— Test, Verification and Optimization. In *IEEE Transactions on CAD*, 1994.

[15] S. Muroga. *Threshold Logic and Its Applications*. New York: WILEY-INTERSCIENCE, 1971.

[16] S. K. Shukla and I. R. Bahar. *Nano, Quantum and Molecular Computing*. KAP, Norwell, MA, USA, 2004.

[17] P. F. Williams, H. Hulgaard, and H. R. Andersen. Boolean Expression Diagram Tool – Version 2.5.

[18] L. Zhang. Threshold Logic Network Synthesis Suite. Master's thesis, Delft University of Technology, 2005.

[19] R. Zhang et al. Threshold Network Synthesis and Optimization and Its Application to Nanotechnologies. In *IEEE Transactions on CAD*, January 2005.