Efficient Synthesis of Approximate Threshold Logic Circuits with an Error Rate Guarantee

Yung-An Lai, Chia-Chun Lin, Chia-Cheng Wu, Yung-Chih Chen[§], Chun-Yao Wang
Department of Computer Science, National Tsing Hua University, Hsincu, Taiwan, R.O.C.
[§]Department of Computer Science and Engineering, Yuan Ze University, Chungli, Taiwan, R.O.C.

Abstract— Recently, Threshold logic attracts a lot of attention due to the advances of its physical implementation and the strong binding to neural networks. Approximate computing is a new design paradigm that focuses on error-tolerant applications, e.g., machine learning or pattern recognition. In this paper, we integrate threshold logic with approximate computing and propose a synthesis algorithm to obtain cost-efficient approximate threshold logic circuits with an error rate guarantee. We conduct experiments on IWLS 2005 benchmarks. The experimental results show that the proposed algorithm can efficiently explore the approximability of each benchmark. For a 5% error rate constraint, the circuit cost can be reduced by up to 65%, and 22.8% on average. Compared with a naive method, our approach has a speedup of 2.42 under a 5% error rate constraint.

I. Introduction

Threshold Logic has been studied since the 1960s. In 1961, an effective method for enumerating threshold functions was proposed [30]. Then, an approach to determining the input weights and threshold value of a Linear Threshold Gate (LTG) was proposed [31]. Later, linear programming methods were proposed to determine whether or not a function could be realized by a single LTG [32]. Despite several achievements on threshold logic in early days, threshold logic did not attract much attention due to lacking efficient methods for hardware implementation.

Recently, with advances of emerging technologies such as Resonant Tunneling Diodes [3], Quantum Celluar Automata [25], Resistant switching devices [12], Spin-based devices [2], and Single Electron Transistor [26], threshold logic becomes more popular than before. Accordingly, the design automation research for threshold logic grows in various aspects including multi-level synthesis methodologies [1][7][11][24][34], static timing analysis [28], verification [20], and testing [13].

On the other hand, threshold logic also strongly binds neural networks [16]. In fact, the neurons in a neural network are all LTGs. Currently, neural networks are the underlying platforms for Artificial Intelligence related applications such as machine learning [18], and pattern recognition [10].

Approximate computing, which is an emerging design paradigm targeting at error-tolerant applications, has been proposed recently. The benefit of approximate computing is to exploit the given error tolerance of applications to implement designs approximately with smaller area, delay, or lower power consumption [8]. Many previous works demonstrated the effectiveness of approximate computing in different design levels ranging from algorithm [9], architecture [17], logic, to transistor levels [14].

Although many approximate logic synthesis techniques have been proposed [6][27][29][33], they are not for threshold logic networks. To directly leverage the inherent error tolerance in the applications of machine learning or data mining that utilize neural networks as platforms, we propose the first work that discusses the approximate logic synthesis for threshold logic networks in this paper.

To quantify errors in approximate computing, two types of error metrics are commonly used [15]: error magnitude and error rate. The error magnitude is usually used in arithmetic circuits to quantify the output differences of correct circuits and approximate ones in numerical values. Conversely, the error rate represents the percentage of all input patterns that produce the erroneous outputs in the approximate circuits. In this work, we adopt the error rate metric, and minimize the synthesized circuits while meeting the error rate constraint.

The main contributions of this work are two-fold:
(1) This is the first work that integrates threshold logic networks with approximate computing.
(2) The proposed approximate operations benefit threshold logic networks optimization with a hybrid cost function.

II. Preliminaries

A. LTG and threshold function

An LTG is a logic gate with n binary inputs and one binary output. The parameters of an LTG are weight w_i, which is corresponding to an input x_i, and the threshold value T. An LTG can be represented by a *weight-threshold vector* $\langle w_1, w_2, ..., w_n; T\rangle$. For example, the LTG in Fig. 1(a) can be represented as $\langle -3, 2, -1, 1; 1\rangle$. The output f of an LTG is evaluated by EQ(1):

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

where w_i and T can be positive or negative integers. In EQ(1), if the weighted summation is greater than or equal to the value T, the output f is 1; otherwise, the output f is 0. For example, an LTG $\langle 2, 1, 1; 3\rangle$ produces 1 under the pattern 110; it produces 0 under the pattern 100. A function that can be realized by a single LTG is called a *threshold function*. A network composed by LTGs is called a *threshold network*, and any functions can be represented by threshold networks.

This work is supported in part by the Ministry of Science and Technology of Taiwan under Grant MOST 106-2221-E-007-111-MY3, MOST 106-2221-E-155-056, MOST 103-2221-E-007-125-MY3

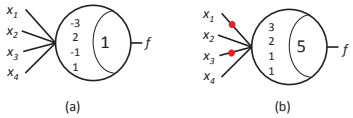
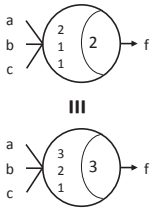


Fig. 1. (a) An LTG having negative weights. (b) The resultant LTG after weight transformation.



a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Fig. 2. The set of CEVs of an LTG.

B. Positive-negative weight transformation

By the definition of an LTG, its weights and threshold value can be positive or negative integers. For ease of analyzing threshold networks, we transform the negative weights into positive ones by the positive-negative weight transformation [23]. The transformation procedure is as follows: (1) Negate the negative weights to positive ones, and add the absolute values of negative weights into the threshold value; (2) Reverse the polarities of corresponding inputs of the negated weights. For example, in Fig. 1(a), -3 and -1 are changed to 3 and 1, respectively. Then, the threshold value 1 is updated as 1+|-3|+|-1|=5. Finally, the polarities of x_1 and x_3 are reversed, represented as dots, as shown in Fig. 1(b). This weight transformation procedure eliminates negative weights without changing the functionality of an LTG.

C. Critical-Effect Vector

For an LTG, there exists at least an input vector, called *Critical-Effect Vector (CEV)*, such that the output changes from 1 to 0 when any one of its inputs in this vector changes from 1 to 0 [22]. For instance, in Fig. 2, (a, b, c) = (0, 1, 1) and (1, 0, 0) are the CEVs of the LTG $\langle 2, 1, 1; 2\rangle$. The complete set of CEVs is an efficient representation for the functionality of an LTG, and is also used to verify the equivalence of two LTGs. Furthermore, when we want to simplify the weights or the threshold value of an LTG without changing its functionality, we can check if the CEVs are intact for determining the validity of weights or threshold value reductions.

D. Hybrid cost function of LTG networks

Gate count is a commonly used metric to evaluate the synthesized results of traditional Boolean networks. For LTG networks, many previous works also used this metric to evaluate the synthesized results [7][20][24]. However, when using the gate count as the cost function, designers tend to obtain a network with a fewer number of LTGs, even the LTGs might have more input variables, larger weights, or larger threshold values. This situation would lead difficulty to or even violate the restrictions in the physical implementation of an LTG [4]. For example, the fanin number of an LTG is generally suggested to be less than seven [23].

On the other hand, some previous work only considered the summation of weights and threshold values as the cost function [22]. This situation may result in a threshold network containing lots of primitive gates e.g. AND, OR, and neglect the compactness characteristic of threshold logic.

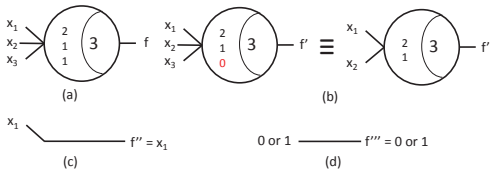


Fig. 3. (a) An original LTG. (b) RW. (c) RG. (d) CC.

In fact, both the gate count and the weight-threshold summation in threshold networks are important as studied in [21], where a hybrid cost function is used as shown in EQ(2).

$$cost = \alpha \cdot \sum_i (\sum_j w_{ij} + T_i) + (1 - \alpha) \cdot |gate| \qquad (2)$$

In EQ(2), α is balance parameter about the gate count and the weight-threshold summation, w_{ij} is the weight of input x_j in gate i, T_i is the threshold value of gate i, and $|gate|$ is the gate count in the whole threshold network. In this work, we also adopt this hybrid cost function in the synthesis of approximate threshold networks.

III. Proposed Method

A. Error rate computation

In the synthesis of approximate circuits, the first issue to be dealt with is error computation. As mentioned, we adopt the error rate as the metric to measure the quality of approximate circuits. To represent the functionality of the original circuit efficiently for further error rate computation, many previous works simulated enormous random vectors with a uniform distribution [33]. These random vectors are saved and regarded as *Golden Vector* (GV) of the original circuit. Then the vectors having different responses at the outputs between the original circuit and the approximate one are called *Erroneous Input Vectors at Output* (EIVO). The ratio of the number of EIVO and the number of GV is defined as the error rate, which is shown in EQ(3).

$$ErrorRate = \frac{|EIVO|}{|GV|} \times 100\% \qquad (3)$$

This paper also adopts the same method to evaluate the error rate of approximate circuits.

B. Approximate operations

We propose three types of approximate operations.

Type 1: Remove one Weight (RW): This operation sets a weight w_i as 0. Then the corresponding input cannot influence this LTG. For example, Figs. 3(a) and 3(b) are an original LTG and the resultant LTG after applying an RW operation on the weight of input x_3. Note that the input x_3 of weight zero can be removed directly as shown on the right of Fig. 3(b).

Type 2: Remove one Gate (RG): This operation removes the LTG and connects one of its input to the output of the LTG. For example, Fig. 3(c) is the resultant circuit (wire) after applying an RG operation. The input x_1 is connected to the output.

Type 3: Change to Constant (CC): This operation changes the LTG to a constant 0 or 1. For example, Fig. 3(d) is a resultant circuit (constant) after applying a CC operation.

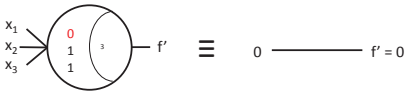


Fig. 4. The special case of the approximate operations.

In our synthesis flow, we will apply these approximate operations to the threshold network repeatedly. For the RW operation, it can be applied to an LTG more than once for further approximation if necessary. For the RG and CC operations, they are not the case. This is because the resultant LTGs become a wire or a constant. However, for a resultant LTG after applying an RW operation, sometimes it would be also simplified as a constant 0. For example, in Fig. 4, the left LTG is evaluated as 0 under all input vectors and is equivalent to a constant 0. Hence, we will check if a resultant LTG is a constant 0 after applying an RW operation. If so, the resultant LTG will be replaced by a constant 0.

C. LTG and approximate operation candidate selection

We apply an approximate operation to a selected LTG repeatedly. A good approximation is to reduce a higher cost while incurring less error. Two candidate selection issues need to be addressed. (1) Which LTG should be selected first for approximation? (2) What kind of operation should be performed on the selected LTG?

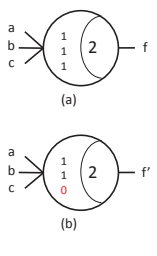
The locations of LTGs in a threshold network show different potential for approximation. If the approximate LTG is near the Primary Inputs (PIs), the error effect caused by approximation might be masked due to more levels in the fanout cone of the LTG. On the other hand, if the approximate LTG is close to the Primary Outputs (POs), more levels in the fanin cone might make the distribution of local inputs occurred in an LTG non-uniform. If the number of input vectors with respect to the erroneous local inputs is fewer, the error rate would be lower. Hence, we observed that different locations for approximation have different benefits to reduce error rates. As a result, the locations of LTGs for approximation are not considered. We randomly choose an LTG for approximation.

However, for the second issue, we have a strategy. Generally, keeping the input with the largest weight in an LTG can approximate the functionality of the LTG with less error. Conversely, removing the input with the smallest weight influences the functionality of the LTG very little. As a result, applying an RG operation with keeping the input of the largest weight, and applying an RW operation with removing the input of the smallest weight are two *preferable* choices in our algorithm.

D. Error rate estimation after approximation

Applying the proposed approximate operations reduces the cost of the threshold network; however, we need to further consider if the error rate exceeds the constraint during approximation. Choosing a candidate LTG with an approximate operation, we have to estimate the corresponding error rate after the approximation. Once a candidate satisfying the error rate constraint is obtained, we accept the candidate and update the current approximate network. The error rate estimation considers two points: (1) Only an overestimation is acceptable. (2) A more accurate estimated result is better.

We divide the discussion into three parts. The same example will be used throughout these three parts. In the example,



a	b	c	f	GVID	CurVID	f′
0	0	0	0	1, 38...	1, 38...	0
0	0	1	0	50, 99...	50, 99...	0
0	1	0	0	11, 67...	11, 67...	0
0	1	1	**1**	28, 55, 69	28, 55, 69	**0**
1	0	0	0	31, 86...	31, 86...	0
1	0	1	**1**	21, 23, 29	21, 23, 29	**0**
1	1	0	1	7, 52...	7, 52...	1
1	1	1	1	29, 30...	29, 30...	1

(c)

Fig. 5. (a) An original LTG. (b) The approximate LTG. (c) Local information at the source.

we approximate the threshold network twice and compute the error rate.

The scenario in this example is set as follows. The total number of golden vectors, $|GV|$, is 100, and the error rate constraint is 4%. To differentiate the EIVO at different iterations, we add an additional superscript on the original notation of EIVO. For example, $EIVO^{(2)}$ stands for the EIVO after the second approximate operations. Note that $EIVO^{(0)} = \emptyset$ due to no approximation.

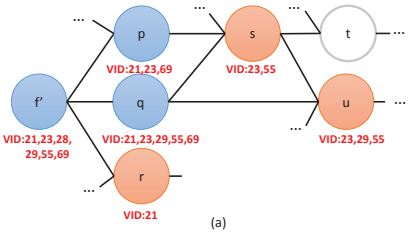
1) Part I: After applying the GV into the original threshold network, we obtain the Local Input Vector (LIV) for each gate. For example, assume that Fig. 5(a) is an LTG f, which will be approximated as f' in Fig. 5(b) by an RW operation. Its truth table containing LIV abc = 000 $\sim$ 111 and the corresponding Vector ID (VID) at the PIs is shown in Fig. 5(c). According to Fig. 5(c), $f \neq f'$ for LIV 011 and 101 and the corresponding VID are 28, 55, 69, 21, 23, and 29. Columns 5 and 6 in Fig. 5(c) represent the Golden Vector ID (GVID) and Current Vector ID (CurVID), respectively. GVID column exhibits the VID distribution in the original threshold network and this information is obtained from GV generation. CurVID column stands for the current VID distribution in the approximate network, and this information will be updated after an approximation is accepted.

The location of the approximate gate f' is called the *source* of this approximation and the corresponding VIDs are called **Erroneous Input Vectors at Source (EIVS)**. Note that since the error effects of EIVS are not always propagated to the POs, we should *not* estimate the error rate as $\frac{|EIVS|}{|GV|} = \frac{|\{28,55,69,21,23,29\}|}{100} = 6\%$, which could be overestimated.

2) Part II: In a global view, the EIVS have considered the influence from the transitive fanin cone of the source. Next, we also need to consider the error effect propagation to the transitive fanout cone of the source. However, if we use the POs as the observation points for checking whether the error effect is propagated out or not, the computation efforts would be large. To balance the computation efforts and the accuracy, we propose a method named *limited-level error simulation*. Before introducing this method, we define a term.

L-Boundary LTGs: Given a source, and a user-defined parameter L, the LTGs in the transitive fanout cone of the source satisfying one of the following conditions are called L-Boundary LTGs.
(1) The shortest level from source to the LTGs is equal to L.
(2) The shortest level from source to the LTGs is less than L, but the LTGs are POs.

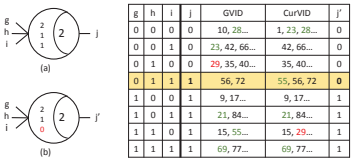
The limited-level error simulation is composed of three steps: (1) Search L-Boundary LTGs in a Breadth First Search



VID	...	21	22	23	24	25	26	27	28	29	...	55	...
EIVB		x		x						x		x	

(b)

Fig. 6. (a) The erroneous status after propagating EIVS of f'. (b) The EIVB of an approximate operation on f'.



g	h	i	j	GVID	CurVID	j'
0	0	0	0	10, 28...	1, 23, 28...	0
0	0	1	0	23, 42, 66...	42, 66...	0
0	1	0	0	29, 35, 40...	35, 40...	0
0	**1**	**1**	**1**	56, 72	55, 56, 72	**0**
1	0	0	1	9, 17...	9, 17...	1
1	0	1	1	21, 84...	21, 84...	1
1	1	0	1	15, 55...	15, 29...	1
1	1	1	1	69, 77...	69, 77...	1

(c)

Fig. 7. (a) An original LTG. (b) The approximate LTG. (c) Local information at the source.

manner. (2) Propagate the error effect in EIVS from the source to the L-Boundary LTGs. The remaining erroneous vectors are called **Erroneous Input Vectors at Boundary (EIVB)**. (3) Estimate the error rate by EQ(4)

$$EstimatedErrorRate = \frac{|EIVO^{(i-1)} \cup EIVB|}{|GV|} \quad (4)$$

where $EIVO^{(i-1)}$ is the EIVO from the last iteration.

For example in Fig. 6(a), the source is f', and we assume that L is set 2 in this example. L-Boundary LTGs are s, u, and r, where r is a PO. Each L-Boundary LTG has its VID. According to Fig. 6(a), we obtain the EIVB = {21, 23, 29, 55}, which is obtained by the union of VID in the L-Boundary LTGs s, u, and r. This result is listed in Fig. 6(b) and also demonstrates that the error effects of some EIVS are blocked. Finally, the error rate is estimated by EQ(4) as $\frac{|\emptyset \cup \{21,23,29,55\}|}{100} = \frac{4}{100} = 4\%$.

The limited-level error simulation reduces the computation effort for estimating the actual error rate. If the estimated error rate is less than the error constraint, the actual error rate will definitely meet the constraint. Hence, we will accept this approximate operation and update the local information of each LTG. On the other hand, if the estimated error rate is larger than the error rate constraint, we will discard this approximation, and evaluate another approximation candidate.

3) Part III: We have already accepted f' for approximation because the estimated result does not exceed the error rate constraint. Furthermore, the updated result shows that $EIVO^{(1)}$ becomes {21, 29, 55}, where VID 23 is eliminated after the updating. Next, we conduct the second approximation for j' using the same method. First, we calculate EIVS of j' by the local information in Fig. 7, and the EIVS are {55, 56, 72}. Second, we conduct the limited-level error simulation, and the EIVB are the same. Finally, the error rate is calculated by EQ(4) as $\frac{|EIVO^{(1)} \cup EIVB|}{|GV|} = \frac{|\{21,29,55\} \cup \{55,56,72\}|}{|GV|} = \frac{|\{21,29,55,56,72\}|}{|GV|} = \frac{5}{100} = 5\%$. Because the error rate exceeds the constraint 4%, the approximate operation on j' is rejected.

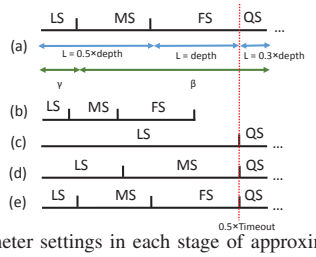


Fig. 8. The parameter settings in each stage of approximation.

In the example of j', we show that some erroneous vectors in the previous approximate operations cannot be included in the EIVB of j'. For example, VID 21 and VID 29 are in the $EIVO^{(1)}$, but they are not seen at the EIVB of j'. Because EIVB of j' only considers the erroneous vectors generated by j', other erroneous vectors might be disappeared in EIVB. However, we have already obtained the overall erroneous vectors from $EIVO^{(i-1)}$ by updating. Hence, the union in EQ(4) combines the error effect of $EIVO^{(i-1)}$ and EIVB such that the estimated error rate will not be underestimated.

E. Approximation heuristic

The proposed approximation heuristic consists of four stages, and they are Leading Stage (LS), Middle Stage (MS), Fine-tuned Stage (FS), and Quick-searching Stage (QS). Two parameters cause the stage transition and they are $AttemptLimit$ and $Timeout$. $AttemptLimit$ is a user-defined parameter for representing the upper bound of the failing trials during approximation. $Timeout$ is a timing budget parameter for the overall approximation. If the selected approximation in one stage violates the error rate constraint for $AttemptLimit$ times consecutively, the heuristic will transit to the next stage. However, for large circuits with more LTGs, when CPU time reaches one-half of the $Timeout$, we jump to QS for converging the approximate results quickly. We discuss the details of the heuristic in the following paragraphs.

We create two queues, High Priority Queue (HPQ) and Low Priority Queue (LPQ), to prioritize the candidate LTGs. We choose the LTGs from the HPQ first. When the HPQ is empty, we choose the LTGs from the LPQ. Initially, we insert every LTG into the HPQ. When an LTG is chosen for approximation but the corresponding estimated error rate exceeds the error rate constraint, a failure counter for that LTG will increase by one. When the failure counter is over a user-defined value, we remove the LTG from the HPQ to the LPQ. Furthermore, if an RG or CC operation is accepted, the corresponding LTG is removed out from the queue immediately.

To achieve a better quality of approximation, we shrink the error rate constraint in the first stage. That is, if the original error rate constraint is β, we set the error rate constraint to be γ in the first stage LS, where $\gamma = 0.25\beta$. The reason behind this is that an approximate candidate consumes most of the error rate budget is not a good candidate. Hence, we exclude this candidate from the result in the LS. For the stages of MS, FS, and QS, we resume the error rate constraint to β.

On the other hand, the limited-level error simulation elevates the efficiency by reducing the search space, but it might sacrifice the quality of the approximation. Generally, a smaller L reduces the estimation effort but could miss some possible candidates. A larger L spends much estimation time for having more accurate estimated results. To tradeoff

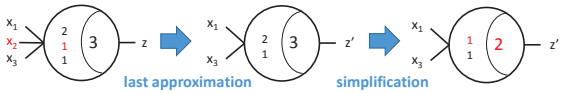


Fig. 9. An example for removing redundant weight and threshold value of an approximated LTG.

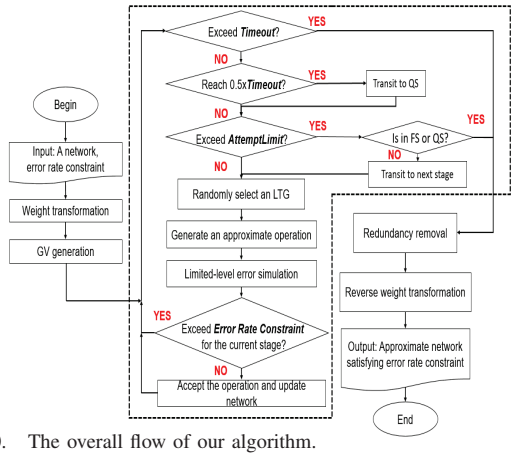


Fig. 10. The overall flow of our algorithm.

estimating effort and approximation quality, we dynamically adjust L in different stages. For the LS and MS, we set L as 0.5×largest depth in the threshold network heuristically. For the FS, we change L to the largest depth to explore the larger solution space, where might not be searched during LS or MS. Conversely, if the heuristic transits to the QS, that means the remaining time budget for approximation is tight, we change L to 0.3×largest depth for further reducing the search space.

We summarize the parameters and stage transitions in Fig. 8. Fig. 8(a) lists the settings under different stages. In Fig. 8(b), it shows a situation that overall CPU time does not exceed one-half of the $Timeout$, and the total CPU time for LS and MS are regarded as a reference for running the FS stage. In Fig. 8(c) $\sim$ Fig. 8(e), the earlier stages reach one-half of the $Timeout$, then they jump to QS stage directly.

F. Redundancy removal

After obtaining a network that cannot be further approximated in our algorithm, we try to remove redundancy on the remaining weights and threshold value for further cost reduction. For example, assume that the original LTG z is in Fig. 9 and z' in the middle of Fig. 9 is an approximate LTG after removing x_2 and the corresponding weight. We can apply the simplification technique proposed in [22] to obtain a more simplified LTG, as shown on the right of Fig. 9.

The simplification technique in [22] is a CEV-based approach. Hence, all of the remaining LTGs are applicable for this simplification technique. We briefly describe two major steps of this simplification technique. (1) The weights are sorted by their values in a decreasing order. (2) Decrease the weights from the largest weight by 1, and the threshold value is also reduced by 1. If the CEVs of the new LTG remain the same as the original LTG, that means the two LTGs are functionally equivalent, the original LTG will be replaced by the new LTG. Otherwise, we repeat the procedure for the next weight. The simplification procedure is terminated until all the weights cannot be further reduced.

G. Overall flow

The overall flow of our algorithm is shown in Fig. 10. The inputs are an initial threshold network and a user-specified error rate constraint. The output is an approximate network satisfying the error rate constraint. We first conduct the weight transformation, followed by the GV generation. Next, the procedures enclosed in a dotted area are the main procedures of our algorithm and divided into two parts. The upper part is for checking $Timeout$ or stage transition. In the lower part, we select LTGs, generate approximate operations, estimate error rate, and update the network repeatedly. When the approximation violates the error rate constraint for $AttemptLimit$ times consecutively in the FS or QS stages, the approximation ends. Next, the weights and threshold values are simplified. Finally, we perform the weight transformation reversely to report the resultant approximate threshold network.

IV. Experimental Results

We implemented our algorithm in C++ language. The experiments were conducted on a 2.6 GHz Linux platform with Intel Xeon E5-2650V2. The benchmarks are from IWLS 2005 [35] and MCNC. Each benchmark was initially synthesized as a threshold network by a tool in [34], using $|PI|$ as the fanin number constraint. The experimental settings were as follows. Since our algorithm involves randomness in some procedures, we conducted experiments on each benchmark for 10 times and report the average results. Note that the experiments on one benchmark with different error rates used the same random seed to ensure the GV are the same for comparison. Furthermore, the size of GV is 10000, which is the same as the previous work [33]. The balance parameter mentioned in Section II-D is set as 0.5. $AttmeptLimit$ is set as 0.7×$|LTG|$ in the network. $Timeout$ is 3600 seconds.

Since this work is the first approximate circuit synthesis targeting at the threshold networks, to demonstrate the effectiveness and efficiency of our approach, we also implemented a naive method, and conducted experiments for comparison. In the naive method, the proposed heuristic in Section III-E was not adopted, i.e., the parameters were fixed for all the stages.

TABLE I shows the experimental results. Column 1 lists the benchmarks. Columns 2$\sim$4 are the initial number of LTGs, weight-threshold summation, and cost in the benchmark. Columns 5$\sim$12 and Columns 14$\sim$21 are experimental results for 5% and 10% error rate constraints. For each error rate constraint, we list the cost, cost improvement (I.), final error rate (E.), and CPU time (T) of the naive method and our approach compared with the initial network. Columns 13 and 22 are the ratios of CPU time between the naive method and our approach for 5% and 10% error rate constraints.

According to TABLE I, the averaged cost improvement of our approach is higher than that of the naive method. Furthermore, our averaged speedups compared against the naive method are 2.42 and 2.14 for 5% and 10% error rate constraints. This is because we do not explore the whole threshold network for estimation during LS, MS, and QS stages. However, for $dalu$, the naive method performed better than ours. This is because $dalu$ has a large depth compared with other benchmarks. The large depth causes the error rate estimation inaccurate in our limited-level error simulation.

TABLE I. THE COMPARISON OF EXPERIMENTAL RESULTS BETWEEN NAIVE AND OUR APPROACH.

				5%									10%								
				Naive				Ours					Naive				Ours				
Bench.	LTG	Sum	Cost	Cost	I.(%)	E.(%)	T	Cost	I.(%)	E.(%)	T	Rat.	Cost	I.(%)	E.(%)	T	Cost	I.(%)	E.(%)	T	Rat.
C1908	287	1332	809.5	516.0	36.26	3.89	249.7	545.3	32.64	4.48	137.0	1.82	565.2	30.18	9.37	225.5	525.0	35.15	9.71	144.7	1.56
usbphy	288	1412	850.0	782.6	7.94	4.95	125.1	765.1	9.99	4.88	72.2	1.73	792.9	6.72	9.95	88.5	747.3	12.09	9.75	76.5	1.16
C1355	340	1678	1009.0	976.8	3.19	4.96	74.0	956.0	5.26	4.96	141.2	0.52	948.5	6.00	9.99	128.9	904.9	10.32	9.95	184.6	0.70
rot	354	1648	1001.0	916.3	8.46	4.99	249.6	884.4	11.65	4.94	135.7	1.84	903.9	9.71	9.99	224.3	852.8	14.81	9.95	154.1	1.46
alu4	372	1774	1073.0	977.3	8.92	4.98	244.9	968.7	9.72	4.93	146.4	1.67	938.2	12.56	9.98	294.7	927.5	13.56	9.96	157.4	1.87
x3	390	1782	1086.0	964.4	11.20	4.98	318.8	938.2	13.61	4.95	110.1	2.90	940.8	13.37	9.97	335.0	906.2	16.56	9.94	124.7	2.69
i2c	504	2966	1735.0	1454.5	16.17	5.00	742.7	1409.3	18.77	4.99	214.1	3.47	1418.7	18.23	10.00	694.5	1351.0	22.13	9.96	248.4	2.80
frg2	509	3069	1789.0	952.4	46.76	4.97	1075.6	946.2	47.11	4.89	435.8	2.47	908.1	49.24	9.89	1005.9	911.1	49.07	9.93	419.6	2.40
pci.	566	2795	1680.5	580.2	65.48	4.99	1007.2	580.8	65.44	4.98	613.8	1.64	531.0	68.40	9.95	959.4	521.9	68.95	9.92	633.6	1.51
simple.	**570**	**2411**	**1490.5**	**1296.4**	**13.03**	**4.99**	**798.8**	**1187.8**	**20.31**	**4.93**	**238.7**	**3.35**	**1295.0**	**13.12**	**9.99**	**750.5**	**1163.9**	**21.92**	**9.91**	**252.4**	**2.97**
pair	712	3394	2053.0	1836.6	10.54	5.00	1743.8	1820.8	11.31	4.95	397.0	4.39	1825.5	11.08	10.00	1562.7	1784.7	13.07	9.92	450.1	3.47
dalu	**782**	**3201**	**1991.5**	**1283.6**	**35.55**	**5.00**	**2462.0**	**1477.0**	**25.83**	**4.99**	**619.5**	**3.97**	**1049.3**	**47.31**	**9.99**	**2348.9**	**1386.7**	**30.37**	**9.98**	**746.5**	**3.15**
C5315	1076	3959	2517.5	2278.1	9.51	4.96	2008.5	2270.8	9.80	4.93	519.0	3.87	2254.1	10.46	9.96	2207.3	2240.5	11.00	9.90	612.3	3.60
s9234	1080	5976	3528.0	2167.3	38.57	5.00	3600.0	1928.2	45.35	4.99	1806.6	1.99	2117.7	39.97	10.00	3600.0	1817.8	48.48	9.98	1981.0	1.82
C7552	1520	5479	3499.5	3141.2	10.24	4.99	3602.0	3121.5	10.80	4.87	1006.9	3.58	3072.1	12.21	9.98	3600.0	3096.6	11.51	9.90	1105.4	3.26
s13207	1943	8397	5170.0	3447.6	33.32	4.98	3600.0	3122.4	39.61	4.99	3597.9	1.00	3412.2	34.00	9.96	3600.0	3110.3	39.84	9.99	3600.0	1.00
system.	2070	9206	5638.0	5475.2	2.89	4.98	3600.0	5054.0	10.36	4.96	3600.0	1.00	5455.7	3.23	9.98	3600.0	5036.3	10.67	9.98	3600.0	1.00
ave.	-	-	-	-	21.06	-	-	-	22.8	-	-	2.42	-	22.69	-	-	-	25.26	-	-	2.14

V. CONCLUSION

This is the first work that integrates threshold networks with approximate computing. We propose three approximate operations for threshold networks. Furthermore, the proposed estimating method tradeoffs accuracy and computing effort, which provides an efficient way to eliminate inappropriate candidate during approximation. According to the experimental results, our approach efficiently reduces the cost of approximate threshold networks.

REFERENCES

[1] L. Amaru' *et al.*, "Majority-based synthesis for nanotechnologies," *Proc. ASP-DAC*, 2016, pp. 499-502.

[2] C. Augustine *et al.*, Low-power functionality enhanced computation architecture using spin-based devices, *Proc. Int. Symp. on Nanoscale Architecture*, 2011, pp. 129-136.

[3] M. J. Avedillo *et al.*, "Multi-Threshold Threshold Logic Circuit Design using Resonant Tunnelling Devices," *Electron. Lett.*, 2003, pp. 1502-1504, vol. 39.

[4] V. Beiu *et al.*, "VLSI Implementations of Threshold Logic-A Comprehensive Survey," *IEEE Trans. on Neural Networks*, 2003, pp. 1217-1243, vol. 14.

[5] V. Beiu, "On Existential and Constructive Neural Complexity Results," *Neural Networks and Computational Intelligence*, 2003, pp. 63-72.

[6] A. Bernasconi *et al.*, "2-SPP Approximate Synthesis for Error Tolerant Applications," *Proc. 17th Euromi. Dig. Syst. Design*, 2014, pp. 411-418.

[7] Y.-C. Chen *et al.*, "Fast Synthesis of Threshold Logic Networks with Optimization," *Proc. ASP-DAC*, 2016, pp. 486-491.

[8] V. Chippa *et al.*, "Analysis and characterization of inherent application resilience for approximate computing," *Proc. DAC*, 2013, pp. 1-9.

[9] V. Chippa *et al.*, "Dynamic effort scaling: Managing the quality-efficiency tradeoff." *Proc. DAC*, 2011. pp. 603-608.

[10] L. O. Chua *et al.*, "Cellular neural networks: applications" *IEEE Trans. on Circuits and Systems*, 1988, pp. 1273-1290.

[11] C.-C. Chung *et al.*, "Majority Logic Circuits Optimisation by Node Merging," *Proc. ASP-DAC*, 2017, pp. 717-719.

[12] D. Goldharber-Gordon *et al.*, Overview of Nanoelectronic Devices, *Proc. IEEE*, 1997, pp. 521-540.

[13] P. Gupta *et al.*, "Automatic Test Generation for Combinational Threshold Logic Networks," *IEEE Trans. CAD*, 2008, pp. 1035-1045, vol. 16.

[14] V. Gupta *et al.*, "IMPACT: imprecise adders for low-power approximate computing" *Proc. ISLPED*, 2011. pp. 409-414.

[15] J. Han *et al.*, "Approximate computing: An emerging paradigm for energy-efficient design," *Proc. ETS*, 2013, pp. 1-6.

[16] G. B. Huang *et al.*, "Can threshold networks be trained directly?," *IEEE Trans. Circuits Syst. II, Exp. Briefs, 53(3)*, 2006, pp. 187-191.

[17] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," *Proc. DATE*, 2016, pp. 1327-1332.

[18] Y. Jin *et al.*, "Pareto-Based Multiobjective Machine Learning: An Overview and Case Studies," *IEEE Trans. on Systems*, 2008, pp. 397-415.

[19] P.-Y. Kuo *et al.*, "On Rewiring and Simplification for Canonicity in Threshold Logic Circuits," *Proc. ICCAD*, 2011, pp. 396-403.

[20] N.-Z. Lee *et al.*, "Analytic approaches to the collapse operation and equivalence verification of threshold logic circuits," *Proc. ICCAD*, 2016, pp. 1-8.

[21] C.-C. Lin *et al.*, "In&Out: Restructuring for Threshold Logic Network Optimization", *Proc. ISQED*, 2017, pp. 413-418.

[22] C.-C. Lin *et al.*, "Rewiring for Threshold Logic Circuit Minimization," *Proc. DATE*, 2014, pp. 1-6.

[23] S. Muroga, "Threshold Logic and its Applications," 1971, New York, NY: John Wiley.

[24] A. Neutzling *et al.*, "Threshold Logic Synthesis Based on Cut Pruning," *Proc. ICCAD*, 2015, pp. 494-499.

[25] M. Perkowski *et al.*, "Logic Synthesis for Regular Fabric Realized in Quantum dot Cellular Automata," *Journal of Multiple-Valued Logic and Soft Comput.*, 2004, pp. 768-773.

[26] V. Saripalli *et al.*, "Energy-delay Performance of Nanoscale Transistors Exhibiting Single Electron Behavior and Associated Logic Circuits," *Journal of Low Power Electronics*, 2010, pp. 415-428,vol. 6.

[27] D. Shin *et al.*, "Approximate logic synthesis for error tolerant applications," *Proc. DATE*, 2010, pp. 957-960.

[28] C.-K. Tsai *et al.*, "Sensitization Criterion for Threshold Logic Circuits and its Application," *Proc. ICCAD*, 2013, pp. 226-233.

[29] S. Venkataramani *et al.*, "Substitute-andsimplify: A unified design paradigm for approximate and quality configurable circuits," *Proc. DATE*, 2013, pp. 796-801.

[30] R. O. Winder, "Single Stage Threshold Logic," *Switching Circuit Theory and Logical Design*, 1961, pp. 321-332.

[31] R. O. Winder, "Threshold Logic," 1962, Ph.D. dissertation, Princeton University, Princeton, NJ.

[32] R. O. Winder, "Enumeration of Seven-Argument Threshold Functions," *IEEE Trans. on Electronic Computers*, 1965, pp. 315-325.

[33] Y. Wu *et al.*, "An efficient method for multi-level approximate logic synthesis under error rate constraint," *Proc. DAC*, 2016. pp. 1-6.

[34] R. Zhang *et al.*, "Threshold Network Synthesis and Optimization and its Application to Nanotechnologies," *IEEE Trans. Comput-Aided Design Integrated Circuits and Systems*, 2005, 24(1):107-18.

[35] http://iwls.org/iwls2005/benchmarks.html