Generating Approximate Boolean Functions

Alexandre R. Crestani[1], Gabriel Ammes[1], Paulo F. Butzen[2] and Renato P. Ribas[1]
[1]*Institute of Informatics and* [2] *School of Engineering*
{*alexandre.crestani, gabriel.ammes, rpribas*}*@inf.ufrgs.br, paulo.butzen@ufrgs.br.*
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil

ABSTRACT

Approximate computing is an emerging design paradigm targeting error-resilient applications. The main objective is to trade the precision of the results for improvements in energy consumption, in the physical size of the circuit, and in the time taken to compute a given task. This paper proposes an approximate logic synthesis approach under error rate and literals reduction rate constraints. The basic idea of our approach is to pick input variables in a Boolean function and shrink them by modifying these variables attributes. We propose two different methods to implement the basic idea. The first algorithm deletes the chosen variable, and the second one turns the variable into a 0 or 1 constant value. We compare the results with the original Boolean function. The experimental results demonstrate that turn the variable into a constant performs better than delete it, obtaining less error rate with higher literals reduction rate.

Keywords: Logic synthesis, Approximate computing

I. INTRODUCTION

In the current world, applications involving digital signal processing, multimedia processing, data analysis, machine learning, and others, are more present in our lives. These applications are what we call *Error-Resilient Applications* [1]. Due to the large and growing use of these applications, it is interesting to develop new tools, methods and techniques to take advantage of this application characteristic.

Approximate computing is a paradigm that trades-off the computational system accuracy by its cost and performance [2]. The use of approximate computing in systems executing error-resilient applications enables the optimization of the system without impacting the quality of the output. The application of approximate computing over the circuit level of a system generates approximate circuits—. The automatic generation of approximate circuit is called *Approximate Logic Synthesis* (ALS) [3]. ALS techniques aim to obtain an approximate circuit for a given original circuit and an error constraint.

There are ALS works that approximate a given circuit by removing input signals of sub-circuits from the original circuit. Wu and Qian [4] present an ALS approach that obtain a two-level expression of sub-circuits from the original circuit and removes literals from this expression to approximate the circuit. Wu et al. [5] propose an ALS approach that removes input signals of sub-circuits of a LUT-mapped circuit to reduce the number of LUTs in the approximate circuit.

In this work, we propose an ALS approach that approximates a circuit represented by a Boolean expression removing the input variable and replacing input variables for constant values. To evaluate the trade-off between accuracy and circuit optimization we obtain the literal reduction and the error rate in the approximate circuit.

The rest of the paper is organized as follows. Section II review the concepts needed to understand and follow the proposed method. Section III explains the proposed method and how it was implemented. Section IV presents and discusses the obtained results. Finally, Section V concludes this paper.

II. BACKGROUND

This section will introduce notation and preliminary concepts necessary to understand this work. It gives the reader a brief about *Boolean Function*, *Logic Trees*, *Error Rate* and *Literal Reduction Rate*.

A **Boolean Function** is a function whose arguments, as well as the function itself, assumes values from a two-element set representing *True* or *False* (usually $\{1, 0\}$) and it takes the form of $F : \{0, 1\}^k \to \{0, 1\}$, where k assume an non negative integer and represents the number of variables in the function.

Therefore, there are 2^k different possible truth tables for a k entries function. It is defined as Truth Table the table used to better comprehend and visualize the Boolean outputs of a given function. Illustration of a truth table for the given example, function F, in Table I.

A Boolean function can be expressed as a directed acyclic graph (DAG). It consists of several nodes representing a logic decision that must be taken in the current analysed variable and two terminal nodes. This structure represents the Boolean result of such combination of variable values and logic operators. Due the similarities with a Binary Tree it is called **Logic Tree**. For example, the following function can be expressed using Logic Tree, as shown in Fig. 1:

Error Rate is the result of dividing the number of wrong cases by the number of possible cases. It is calculated by $A/2^k$, where A is the number of different outputs between two truth tables and n is the function's number of inputs.

Literal Reduction Rate is the amount of literals remained from a modified function (A) compared to the amount of literals counted on the original one (B). This metric is important to better visualize the size of the modified circuit compared to the original one. It is calculated by $1 - A/B$.

TABLE I
TRUTH TABLE OF THE FUNCTION $F(a,b,c) = \overline{a} * \overline{b} * \overline{c} + b * (a+c)$

a	b	c	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

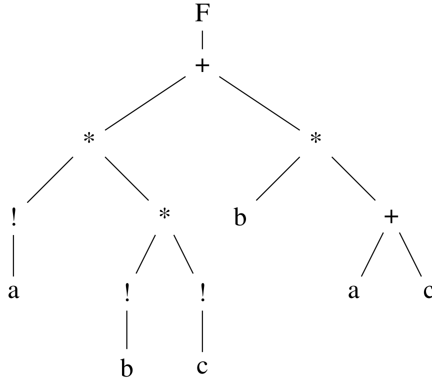

Fig. 1. Logic Tree representation of F function.

III. PROPOSED METHODS

ALS explores the concept of generating acceptable errors in Boolean Function to improve circuit metrics as area, power and performance. We investigate deleting/fixing a variable in terms of literal count reduction and error insertion. Our method changes a variable for a constant value or removes a variable and then calculates the number of literals of approximate function and the error rate.

We implemented the above presented ideas. The implemented algorithm consists in, given a Boolean expression in a String format, approximate it and calculates the Error Rate and the approximated Boolean function. To do it the software converts the Boolean expression into a Logic Tree. Then, it operated the Logic Tree to approximate it and to obtain the number of literals and minterms of the approximated Boolean function.

The main approaches used to realize our approximation are:

- **Constant Substitution**: Replace all logic tree leaves that contains the given variable for a constant 0 or 1.
- **Variable Removal**: Removes all leaves containing the given variable.
- **Constant Propagator**: Simplify the logic tree as much as possible after modify the Boolean function.
- **Dangling Swept**: Modify the logic tree with nodes with less than the minimum number of inputs.
- **Literals Count**: Returns the number of literals (leaves) of a logic tree.
- **Get Minterms**: Returns the minterms covered by the Boolean function represented by the logic tree.

A. Constant Approximation

To perform the constant substitution approximation, it is applied the constant substitution approach to change all variables occurrences on the leaves of the logic tree for a constant value. Than it is necessary to simplify the logic tree considering these constants leaves. The constant propagation approach was implemented to do these simplifications. The modifications on the logic tree are based on the Boolean equivalences presented in Table II.

TABLE II
BOOLEAN EQUIVALENCES.

Original	Equivalent
$a * 0$	0
$a * 1$	a
$a + 0$	a
$a + 1$	1

Lets analyse the behavior of the Logic Tree class doing some simplifications after receives the command to change the variable *c* for the constant value *1*. Modified Boolean function representation below, at Fig. 2.

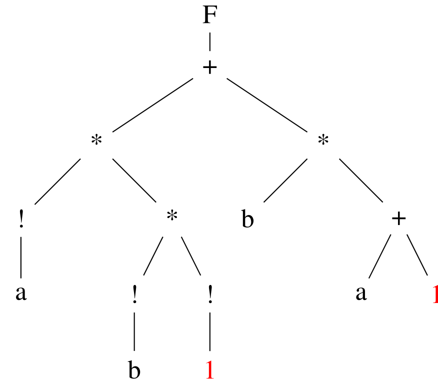

Fig. 2. Logic Tree of F changing *c* for *1*.

In this particular case the first thing it does is simplify the $\overline{b} * \overline{1}$ and the $a+1$ factors. Reaching the representation bellow, at Fig. 3.

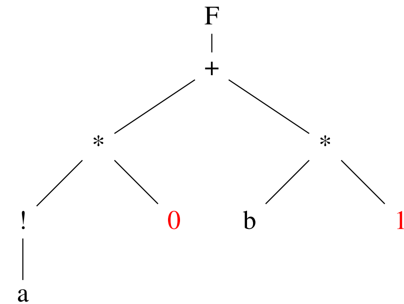

Fig. 3. Logic Tree of F changing *c* for *1*. Step 1 of the simplification.

As the logic tree still have more simplifications available, it does it again and again until it get as simple as possible. The next step is to simplify the $\overline{a} * 0$ and $b * 1$, getting Fig 4.

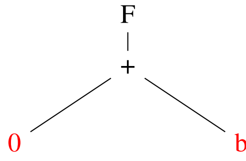

Fig. 4. Logic Tree of F changing c for *1*. Step 2 of the simplification

Then it does the final simplification for the current example. Simplify the $0+b$, getting its final form as represented at Fig. 5.

F

Fig. 5. Logic Tree of F changing c for *1*. Step 3 of the simplification

Therefore, given $F(a,b,c) = \overline{a}*\overline{b}*\overline{c}+b*(a+c)$, the target variable c and by following the steps above we get the follow function by turning this variable into 1:

$$F_1(a,b,c) = b$$

B. Variable Removal

The Removal Variable procedure is based on the change variable steps previously presented. However, instead of set the value of the nodes which once contained the target variable, now these nodes are excluded from the Boolean Function.

Note that when a variable is deleted, the logical operator which once connected this variable to another variable or sub-tree is also deleted. Therefore, excluding a variable is the same as replace the operators node which connects the variable to a sub-tree for this sub-tree.

To illustrate the procedure, lets consider the same Boolean Function used before and delete the variable c. In Fig. 6 we can see the illustration of the nodes affected by the variable to be deleted. Fig. 7 shows the Logic Tree after in fact delete the variable.

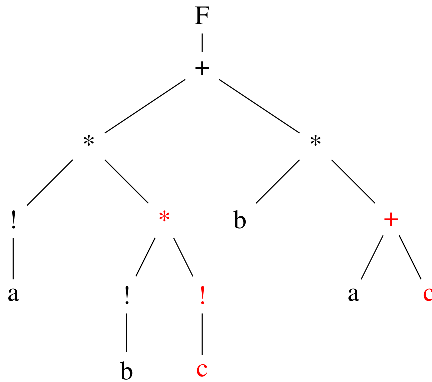

Fig. 6. Logic Tree of F with the nodes affected by deleting c highlighted.

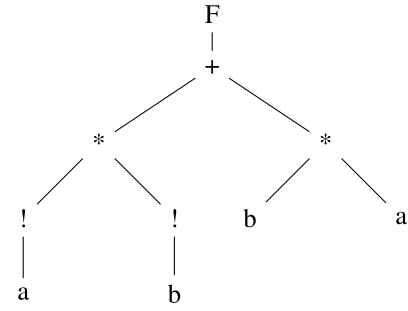

Fig. 7. Logic Tree of F after deleting c.

Therefore, given the original Boolean function $F(a,b,c) = \overline{a}*\overline{b}*\overline{c}+b*(a+c)$, the target variable c and following the simplification steps, we get

$$F_r(a,b,c) = \overline{a}*\overline{b}+b*a$$

C. Error Rate Calculation

The Error Rate is computed dividing the difference between the True outputs from the original Boolean function and the modified ones by the total number of possible outputs from the given Boolean Function.

Let A be a set containing minterms that are True outputs on the original function. Let B a set containing all minterms that are True outputs on the modified function. Be n the number of variables of this function. The Error Rate is calculated by the following steps:

1) Set the number of elements in $(A-B \cup B-A)$ as DIFF.
2) Set the number of total possible outputs (2^n) as TOTAL.
3) Calculate the Error Rate doing $DIFF/TOTAL$

The example bellow illustrates the previous steps. We compare and calculate the error of the functions (original and modified ones) previously used in this work.

TABLE III
TRUTH TABLE OF THE ORIGINAL FUNCTION
$(F(a,b,c) = \overline{a}*\overline{b}*\overline{c}+b*(a+c)$

a	b	c	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Now we can clearly see the difference between the outputs and the function representation itself of the original and the modified function.

Let:

- ER_1 be the error rate comparing F and F_1.
- A be the set of outputs in F.
- B_1 be the set of outputs in F_1.

Then, using all Error Rate Calculations concepts defined in this subsection, we calculate:

TABLE IV
TRUTH TABLE OF THE MODIFIED FUNCTION $F_1(a, b, c) = b$

a	b	c	F
0	0	1	0
0	0	1	0
0	1	1	1
0	1	1	1
1	0	1	0
1	0	1	0
1	1	1	1
1	1	1	1

$ER_1 = |(A - B_1 \cup B_1 - A)|/2^n$
$= |([0,3,6,7] - [2,3,6,7] \cup [2,3,6,7] - [0,3,6,7])|/2^3$
$= |([0] \cup [2])|/8 = 2/8 = \mathbf{25\%}$

To summarize, error rate is the amount of different outputs divided by the amount of possible outputs, as shown above. In this case the method returned a 25% error rate.

IV. RESULTS

All three described operations were applied to each variable belonging to each analyzed Boolean function. The implemented software was run once for each of these cases, generating data for an analysis. In that way, the computational cost grows with the function's number of inputs.

The idea is to get not just the *Error Rate* that occurs for each case, but also get the amount of literals that appears in those cases. With these data we can also calculate the *Literals Reduction Rate*, as already explained before. We applied our method over 8 function: four generic functions, the functions of each output of the C17 circuit from ISCAS'85 benchmarks, the function of the carry chain of a Carry-lookahead adder and the function of the Z9sym circuit from IWLS'93 benchmarks.

The Table V summarizes the obtained results. This table is divided in nine sections, as follows.

The first column keeps original function. The second column, the target variable. From the third to the fifth, the error rate generated by, respectively, delete, turn to 0 and turn to 1. The sixth column keeps the original amount of literals. From the seventh to the ninth keeps the amount of literals on the modified functions keeping the same order from third to fifth.

From the presented results, it is possible to note that deleting a variable lead to worst results when compared to the solutions that fixed it in 0 or 1. The solutions that set the variable as 0 or 1 always shows the same error rate. The difference between them can be observed in the literal reduction.

As introduced, same error rates can occurs when applying each procedure over the same function and variable, but the amount of literals can be different. As example, for the third function, meanwhile the *remove variable* method returned a function with four literals, the *change to 1* method returned a function with only 1 literal. Respectively, this represents a Literals Reduction Rate of 33.33% and 83.33%. It shows us that, for the variable *c* in target of this Boolean function, the *change to 1* method is better than the *delete variable* method.

V. CONCLUSIONS

In this work we investigates the efficiency of approximating Boolean functions generation in terms of error rate and literal reduction considering three different strategies. An exhaustive analysis were performed in some benchmarks functions and the results indicate the strategies of fixing the variables to 0 or 1 as the best alternatives.

ACKNOWLEDGMENT

This study was financed by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

TABLE V
RESULTS OF APPLYING THE THREE PROPOSED PROCEDURES FOR EACH VARIABLE OF THE AIMED FUNCTIONS.

Function	Var	Error Rate			Literals			
		rem	to 0	to 1	orig	rem	to 0	to 1
$!(a * b)$	a	25,0%	25,0%	25,0%	2	1	0	1
	b	25,0%	25,0%	25,0%	2	1	0	1
$!(a * b * c)$	a	12,5%	12,5%	12,5%	3	2	0	2
	b	12,5%	12,5%	12,5%	3	2	0	2
	c	12,5%	12,5%	12,5%	3	2	0	2
$!a*!b*!c+$ $b*(a+c)$	a	25,0%	25,0%	25,0%	6	4	4	1
	b	50,0%	50,0%	50,0%	6	4	2	2
	c	25,0%	25,0%	25,0%	6	4	4	1
$!(a+b+c+d)$	a	6,2%	6,2%	6,2%	4	3	3	0
	b	6,2%	6,2%	6,2%	4	3	3	0
	c	6,2%	6,2%	6,2%	4	3	3	0
	d	6,2%	6,2%	6,2%	4	3	3	0
$C17y_0$ $!(!(a*c)*$ $!(b*!(c*d)))$	a	18,7%	18,7%	18,7%	5	4	3	4
	b	31,2%	31,2%	31,2%	5	4	2	3
	c	18,7%	18,7%	18,7%	5	3	1	4
	d	6,2%	6,2%	6,2%	5	4	3	4
$C17y_1$ $!(!(a*!(b*c))*$ $!(d*!(b*c)))$	a	18,7%	18,7%	18,7%	6	5	3	5
	b	18,7%	18,7%	18,7%	6	4	2	4
	c	18,7%	18,7%	18,7%	6	4	2	4
	d	18,7%	18,7%	18,7%	6	5	3	5
CLA Carry	C0	0,2%	0,2%	0,2%	15	14	10	11
	G0	0,6%	0,6%	0,6%	15	14	11	12
	G1	2,1%	2,1%	2,1%	15	14	12	13
	G2	8,4%	8,4%	8,4%	15	14	13	14
	G3	33,3%	33,3%	33,3%	15	14	14	0
	P0	0,2%	0,2%	0,2%	15	14	10	14
	P1	0,9%	0,9%	0,9%	15	13	6	14
	P2	4,1%	4,1%	4,1%	15	12	3	14
	P3	16,6%	16,6%	16,6%	15	11	1	14
Z9sym	a	19,1%	10,9%	10,9%	334	296	208	206
	b	12,1%	10,9%	10,9%	334	287	217	210
	c	11,3%	10,9%	10,9%	334	302	200	238
	d	10,9%	10,9%	10,9%	334	326	208	253
	e	10,9%	10,9%	10,9%	334	330	202	167
	f	10,9%	10,9%	10,9%	334	306	156	156
	g	15,2%	10,9%	10,9%	334	278	177	188
	h	17,1%	10,9%	10,9%	334	269	204	178
	i	15,2%	10,9%	10,9%	334	278	219	179

REFERENCES

[1] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing." in *2013 Design Automation Conference (DAC)*, 2013.
[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, 2016.
[3] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.
[4] Y. Wu and W. Qian, "Alfans: Multilevel approximate logic synthesis framework by approximate node simplification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1470 – 1483, 2020.
[5] Y. Wu, C. Shen, Y. Jia, and W. Qian, "Approximate logic synthesis for FPGA by wire removal and local function change." in *2017 Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.