DPALS: A Dynamic Programming-based Algorithm for Two-level Approximate Logic Synthesis

Chen Zou[1], Weikang Qian[2], Jie Han[3]
[1] The State Key Lab of ASIC & System, Fudan University, Shanghai 200433, China
[2] UM-SJTU Joint Institute, Shanghai Jiao Tong University, Shanghai, 200240, China
[3] Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, T6G1H9, Canada
Email: [1]czou12@fudan.edu.cn, [2]qianwk@sjtu.edu.cn, [3]jhan8@ualberta.ca

Abstract

Approximate circuit design is an emerging paradigm in which a designer deliberately changes the specified Boolean function to reduce area, delay, and/or power consumption of a circuit. This paper focuses on the synthesis of approximate logic circuits (or ALS) under a given error constraint. In particular, we consider ALS for a two-level design under an error rate constraint. A dynamic programming-based algorithm is proposed to find a nearly optimal approximate function by identifying the most promising set of cubes to be added to the on-set of the original function. Then, an off-the-shelf two-level logic synthesis tool is applied to further optimize the sum-of-product (SOP) expression. The experimental results show that the literal reduction is close to the optimal solution when the error rate constraint is tight and that more than 50% literal reduction is achieved for error rate below 0.8% for an 8-bit adder and a square root circuit.

1. Introduction

With the continued scaling of CMOS devices, the reduction of power consumption becomes one of the key issues for integrated circuit design. Recently, a new paradigm of approximate circuit design was proposed for applications that can tolerate error, including image processing, machine learning, and pattern recognition [1]. An approximate circuit requires a smaller area, shorter delay, and/or lower power consumption than the circuit that implements the correct function.

Most prior work has focused on ad-hoc approximate circuit designs. This includes arithmetic circuits of adders [2] and multipliers [3]. Recently, several efforts have been made on automatically synthesizing approximate circuits, known as the approximate logic synthesis (ALS). In [4], a method was proposed by expanding the existing prime implicant (PI) in a 2-level circuit. It uses maximal allowable error rate (ER) as the error constraint. The work in [5] also focused on 2-level ALS, but it considered a more general error constraint that includes both the error rate and the error magnitude (EM). The EM constraint was handled by reducing the problem to minimizing the Boolean relation (BR), which was further solved using an efficient BR-solver. The authors in [6] proposed an ALS algorithm for multi-level circuits. It injects stuck-at faults into the original circuit, followed by redundancy removal. In [7], another ALS algorithm for a multi-level design was proposed. It introduced a quality constraint circuit, which helps convert the ALS problem into a conventional logic synthesis problem.

In this paper, we focus on the two-level ALS problem constrained by ER. A dynamic programming-based ALS algorithm, DPALS, is proposed to find a nearly optimal approximate function by identifying the most promising set of cubes to be added to the on-set of the original function. Albeit introducing errors, the added cubes make many cubes in the original sum-of-product (SOP) expression redundant and enable the expansion of many cubes. As a result, the added cubes will lead to a large reduction in the number of literals for the SOP expression, which is a good indicator of the area and power consumption of a two-level circuit. The approximate function can be further minimized by an off-the-shelf two-level logic synthesis tool such as Espresso [8].

The contributions of this work are listed as follows:

(1) To efficiently search for cubes to be added into the original function, we propose to handle the problem on the Hasse diagram of all cubes in a Boolean space.

(2) A dynamic programming-based efficient algorithm is proposed to work on the Hasse diagram for ALS.

(3) A partitioning technique is proposed to handle large circuits. This technique enables a parallel execution on multi-core CPUs.

2. Preliminaries on Logic Synthesis

An m-variable Boolean space is denoted as B^m, where $B = \{0,1\}$, and there are m variables in B^m as $x_1, x_2, \ldots, x_m$. For a variable x, x and $\bar{x}$ are called *literals*. A *cube* is a conjunction of literals such that x and $\bar{x}$ do not appear simultaneously. A *minterm* is a cube in which each of the m variables appears once, in either its original or complemented form.

The *on-set* (*off-set*) of a single-output Boolean function is the set of minterms that let the function evaluate to 1 (0). A cube *belongs to* a Boolean function if and only if all the minterms contained in the cube belong to the on-set of the function. If a cube belongs to a Boolean function, the Boolean function *covers* the cube.

3. Proposed Algorithm

In this section, the main algorithm, DPALS, is presented. For convenience, we illustrate the algorithm on single-output functions. For a multi-output function, we can divide it into multiple single-output functions, then apply the proposed DPALS algorithm on each individual function, and finally combine the results.

3.1 Problem Formulation

The problem we consider here is the ALS constrained by an upper bound on error rate. It is formally defined as follows:

Given an original function *f* and an error rate *ER*, we want to find an SOP expression with the least number of literals such that the number of input patterns of the SOP that produces a different output than *f* is no more than $ER \cdot 2^m$, where *m* is the number of inputs of *f*.

The problem is solved in two steps in our proposed algorithm. The first step is to apply DPALS to find an approximate function f_{Apx}, and the second step is to call a two-level logic synthesis tool (such as Espresso) to minimize the SOP expression of f_{Apx}.

3.2 Basic Ideas of DPALS

We illustrate the basic idea behind DPALS through an example. Consider a function $h = \overline{x_1}\overline{x_2}x_4 + \overline{x_1}x_2\overline{x_3} + \overline{x_1}x_3\overline{x_4} + x_1x_2x_4 + x_1x_3x_4$, the k-map of which is shown in the left of Fig. 1. As we can see, the minimized SOP expression of the original function contains five small cubes. However, if we add a big cube $\overline{x_1}$ (corresponding to the green circle in the k-map) to the on-set of the function, then, despite introducing errors, this new cube could make three original cubes $\overline{x_1}\overline{x_2}x_4, \overline{x_1}x_2\overline{x_3}, \overline{x_1}x_3\overline{x_4}$ redundant and remove the literal x_1 in an original cube $x_1x_2x_4$, because the cube can be expanded to the cube x_2x_4.

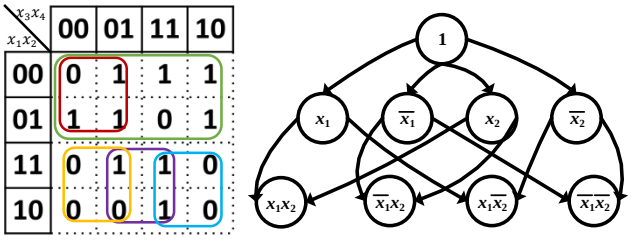

Figure 1. Left: the k-map of a function *h*; Right: the Hasse diagram of all the cubes in the Boolean space B^2.

Thus, the basic idea of DPALS is to add cubes that do not belong to the onset of original function to achieve literal reduction, just as the example shown above. The difficulty lies in how to evaluate different sets of cubes and find the most promising set. To address this challenge, we introduce two metrics for a set of cubes, the number of errors (NE) and the performance vector (PV). The first metric measures the number of input patterns that will have error when the set of cubes is added and the second metric estimates the impact on literal reduction. They are discussed in detail in Sections 3.4 and 3.5, respectively. The calculation of NE and the selection of cubes are performed on a Hasse diagram of all cubes, which is discussed in the next subsection.

3.3 Hasse Diagram of All Cubes in a Boolean Space

The Boolean space B^m contains 3^m cubes. We organize the cubes as a Hasse diagram to facilitate the calculation of NE and execution of the dynamic programming algorithm. Hasse diagram is a directed acyclic graph and is used to represent a set that has a partial ordering. For the complete set of cubes in a Boolean space, we can define a partial ordering on the cubes as follows: a cube c_1 precedes another cube c_2 if and only if cube c_1 covers cube c_2. An example of the Hasse diagram of all the 9 cubes in B^2 is shown in the right of Fig. 1. Each edge connects a parent cube to one of its child cubes. For a parent cube with *l* variables absent, it has *l* pairs of child cubes. The *i*th pair of child cubes can be obtained by multiplying the parent cube with the *i*th absent variable in both its complemented and original forms. Note that two child cubes in a pair do not share any minterm, and the combination of them results in their parent cube.

3.4 Number of Errors in a Set of Cubes

The number of errors (NE) in a set of cubes is defined as the number of input patterns that have output error when the set of cubes is added into the original function. It's equal to the number of minterms in the set of cubes that belong to the off-set of the original function.

In order to calculate the NEs for an arbitrary set of cubes, we first obtain the NEs for all individual cubes. They are calculated level by level in the Hasse diagram from its bottom level to its top level. We first compute the NEs of all cubes at the bottom level of the Hasse diagram, i.e., the minterms. If a minterm belongs to the off-set of the original function, then its NE is 1. Otherwise, if the minterm belongs to the on-set or the don't care set, its NE is 0. The NE of a cube c in a higher level is obtained as the sum of NEs of a pair of child cubes of *c*. For example,

$$NE[x_2\overline{x_4}] = NE[\overline{x_1}x_2\overline{x_4}] + NE[x_1x_2\overline{x_4}].$$

3.5 Performance Vector of a Set of Cubes

The metric performance vector (PV) of a set of cubes estimates the impact on literal reduction when the set is added into the original function. Let us use the function *h* with its k-map shown in Fig. 1 as an example again to investigate how to define this metric. Assume that the upper bound on error rate is 1/8. This means that the approximate Boolean function could have at most two input patterns producing wrong outputs. We could have the following three choices to add a set of cubes which causes two input patterns to produce wrong outputs: (1) adding the blue cube only, (2) adding the green cube only, and (3) adding both the red cube and the purple cube. As we can see, choice (1) only makes one cube redundant for the original function, while the other two choices make 3 original cubes redundant and expand the other 2 original cubes. Notice that the set of cubes of choice (1) contains 4 minterms while the sets of cubes of the other two choices both contain 8 minterms. Thus, the more minterms contained by a set of added cubes, the bigger the literal reduction could be. Furthermore, by comparing choice (2) and choice (3), we find that the choice (2) is better, since it just adds one big cube instead of two small cubes.

Based on the above observation and the tests on some

randomly generated functions, we propose to use a vector (p, q) as PV, where p is the total number of minterms contained in the set of cubes that estimates the likelihood to cover or expand original cubes, and q is the total number of literals in the added cubes that measures the overhead of adding this set to the original function. For example, the PVs of the sets of cubes in choices (1), (2), and (3) are (4, 2), (8, 1), and (8, 4), respectively. From the above example, we can see that a set of cubes can reduce more literals when added into the original SOP if it contains more minterms or has smaller literal overhead. Thus, given two PVs (p_1, q_1) and (p_2, q_2), we define $(p_1, q_1) > (p_2, q_2)$ if and only if either $p_1 > p_2$, or $p_1 = p_2$ and $q_1 < q_2$.

3.6 Dynamic Programming for Finding the Best Set of Cubes

With the definition of NE and PV, the way we propose to find a nearly optimal approximate function is to find the set of cubes with the highest PV among all the sets with NE no more than $L = ER \cdot 2^m$, where m is the number of inputs. This set is then added to the original function to construct an approximate function. A dynamic programming algorithm carried out on the Hasse diagram of all cubes is used to find the best set of cubes.

For each cube c_i in the Hasse diagram and each $0 \leq j \leq L$, we first define a *candidate set of cubes* for c_i and j as a set of descendent cubes of c_i in the Hasse diagram satisfying that none of the cubes belong to the original function and the NE of the set of cubes equals to j. We then define a general problem $Q[c_i][j]$ to find a set with the highest PV among all candidate sets of cubes for c_i and j. The highest PV is denoted as $g[c_i][j]$. Note that the original problem of finding the best set of cubes to be added to the original function is equivalent to finding the best solution among solutions to $Q[1][0]$, ... , $Q[1][L]$, where 1 in the first pair of brackets refers to the root cube in the Hasse diagram.

First, we show g for two base cases:

(1) The case where $j = 0$, which means that the NE of a candidate set of cubes is 0. However, this contradicts the requirement that the cubes in the candidate set do not belong to the original function. Thus, no candidate sets of cubes exist for this case, i.e. $g[c_i][0]$=(0, 0).

(2) The case where c_i is a minterm. c_i only has one descendent, which is itself. Thus, we only need to consider the cube set $\{c_i\}$. $\{c_i\}$ is a candidate set for c_i and j only when $NE[c_i] = 1$ and $j = 1$, because only in this situation does c_i not belong to the original function and $NE[c_i] = j$. In this case, the best set for the problem $Q[c_i][j]$ is $\{c_i\}$ and $g[c_i][j] = (1, m)$. When $j \neq 1$ or $NE[c_i] \neq 1$, there are no candidate set of cubes for c_i and j. Thus, $g[c_i][1] = (0, 0)$.

Next we characterize the optimal structure of the problem $Q[c_i][j]$ for general cubes c_i and $j > 0$. We assume that c_i has $l \geq 1$ absent variables.

A candidate set S of cubes for c_i and j can be constructed by the union of a candidate set S_0 of cubes for $c_{r,0}(c_i)$ and k and a candidate set S_1 of cubes for $c_{r,1}(c_i)$ and $(j - k)$, where $c_{r,0}(c_i)$ and $c_{r,1}(c_i)$ denote the rth pair of child cubes of c_i and $0 \leq k \leq j$. Since the sets S_0 and S_1 are composed of descendent cubes of $c_{r,0}(c_i)$ and $c_{r,1}(c_i)$, respectively, they are disjoint. Thus, we can calculate the PV of set S as the sum of the PVs of S_0 and S_1, since the number of minterms and the literal overhead are added up when combining two disjoint sets of cubes. Therefore, the optimal $g[c_i][j]$ should be at least

$$M_1 = \max_{\substack{1 \leq r \leq l \\ 0 \leq k \leq j}} (g[c_{r,0}(c_i)][k] + g[c_{r,1}(c_i)][j - k]). \quad (1)$$

For those pairs of c_i and j such that $NE[c_i] = j$, there is an extra candidate set, $\{c_i\}$, of which the PV is

$$M_2 = (2^l, m - l). \quad (2)$$

Thus, the highest PV $g[c_i][j]$ for more general cube c_i and $j > 0$ can be calculated as

$$g[c_i][j] = \begin{cases} \max\{M_1, M_2\} & if\ NE[c_i] = j \\ M_1 & if\ NE[c_i] \neq j \end{cases} \quad (3)$$

As shown by (3), g for a parent cube can be obtained from g for the child cubes. Thus, we can obtain g for the root cube of the Hasse diagram in a bottom up way starting from the last level of cubes in the Hasse diagram. With the best sets of cubes for the root cube and different NEs obtained, the set of cubes with the highest PV among those best sets will be chosen to be added to the on-set of the original function to form the approximate function. Then, Espresso is utilized to further obtain a minimal SOP expression. The time and the space complexity of DPALS are $O(3^m \cdot L^2)$ and $O(3^m \cdot L)$, respectively, where m is the number of inputs and L is the upper bound of number of incorrect input patterns.

4. Partitions of Large Circuits

Since the size of the Hasse diagram increases exponentially with the number of inputs, there will be a memory overflow for functions with a large number of inputs. To solve this problem, we propose to partition a complex function into several simpler functions and apply DPALS to each of them. A parameter M is introduced to control the maximum memory usage. The function will be divided according to the combination of the first $(m - M)$ variables. For example, if m=12 and M=10, we will split the original function into 4 simpler functions which produce the original output for the input combination 00----------, 01----------, 10----------, and 11----------, respectively. Indeed, these four functions correspond to the four *cofactors* of the original function with respect to x_1 and x_2. Then DPALS is applied to these functions under the same error rate constraint. By combining all approximate functions for these functions, we obtain a nearly optimal approximation for the original function.

With the partition, the space complexity becomes independent of the number of inputs at the cost of some quality degradation. The parameter M for controlling the memory usage is determined empirically by sweeping M

from 7 to 12 on 12 randomly generated Boolean functions and observing tradeoffs on literal reduction with runtime and memory usage. The value of *M* we finally choose is 11. Note that this partition method can also enable a parallel execution that further reduces the total runtime significantly.

5. Experimental Results

All the experiments are carried out on a PC with a quad core I7-3610QM CPU with 4GB RAM.

5.1 Optimality Study of DPALS

In this section, DPALS are tested on simple functions in order to compare its results with the optimal results produced by an exhaustive search (EXS). The comparison results are shown in Table 1. The first column of the table shows the circuit/function name, the number of inputs (*i*), and the number of outputs (*o*). The two ER constraints for different circuits shown in the second column correspond to the number of incorrect input patterns of 1 and 2, respectively. As shown, DPALS produces results very close to the optimal ones for most of these circuits.

5.2 Results of DPALS on Large Circuits

In this section, DPALS combining the partition method is applied to an 8-bit adder, a square root circuit, and a multiplier circuit under the ER constraints of 0.8%, 1.6% and 2.4%. The number of literals in the final SOP expression and the runtime are shown in Table 2. The first column of the table shows the name of the original function, the number of inputs (i), the number of outputs (o), and the number of literals in the minimized SOP expression of the original function (L). We also provide the runtime of the parallel execution (P_Runtime) for which DPALS is applied to different partitions in parallel by fully utilizing the four cores in the CPU. As shown, DPALS achieves more than 50% literal reduction even for a tight ER constraint of 0.8% for the adder and the square root circuits. Parallel computing significantly reduces the runtime of DPALS.

6. Conclusion

In this paper, a novel algorithm is proposed for approximate logic synthesis (ALS). It identifies the most promising set of cubes to be added to the original function to produce a nearly optimal approximate function. It utilizes dynamic programming on the Hasse diagram of all cubes. Experimental results have shown the effectiveness and efficiency of the proposed algorithm.

Acknowledgements

Chen Zou thanks China Scholarship Council (CSC) for the funding provided for him to participate a summer research intern program at the University of Alberta. This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61472243 and 61574089, and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Table 1. Comparison between the results from DPALS and the optimal results found through exhaustive search (EXS).

Two-level Circuit	ER	Number of literals			Time (s)	
		Original	EXS	DPALS	EXS	DPALS
z9sym i:9; o:1	2‰	516	497	497	4.743	0.071
	4‰	516	474	474	160.760	0.096
sym10 i:10; o:1	1‰	1260	1134	1140	15.867	0.132
	2‰	1260	1080	1107	872.001	0.142
rd73 i:7; o:3	2.6‰	756	735	743	5.328	0.040
	5.2‰	756	720	734	402.723	0.040
Clip i:9; o:5	0.4‰	631	601	620	62.759	0.132
	0.8‰	631	592	611	208min	0.120
sao2 i:10; o:4	0.25‰	421	420	421	108.674	0.192
	0.5‰	421	418	421	430min	0.202

Table 2. Experimental results of DPALS on large circuits.

ER		0.80%	1.60%	2.40%
add8 i:16; o:9 L:11972	Literal	5922	4225	3073
	Reduction %	50.53%	64.71%	74.33%
	Runtime (s)	245.32	383.53	398.77
	P_Runtime (s)	67.673	95.508	108.47
mul8 i:16; o:16 L: 360494	Literal	303828	281321	262342
	Reduction %	15.72%	21.96%	27.23%
	Runtime (s)	465.12	742.11	778.37
	P_Runtime (s)	128.49	200.42	214.43
root8 i:16; o:8 L:11301	Literal	5507	5098	4654
	Reduction %	51.27%	54.89%	58.82%
	Runtime (s)	114.2	173.53	180.43
	P_Runtime (s)	32.74	48.662	51.45

References

[1] Han J., Orshansky M., Approximate computing: an emerging paradigm for energy-efficient design, ETS 2013, pp.1 (2013)

[2] Miao J., He K., Gerstlauer A., and Orshansky M., Modeling and synthesis of quality-energy optimal approximate adders, ICCAD 2012, pp.728 (2012)

[3] Liu C., Han J. and F. Lombardi, A low-power, high-performance approxiamte multiplier with configurable partial error recovery, DATE 2014, pp. 95 (2014)

[4] Shin D., and Gupta S.K., Approximate logic synthesis for error tolerant applications, DATE 2010, pp.957 (2010)

[5] Miao J., He K., Gerstlauer A., and Orshansky M., Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints, ICCAD 2013, pp.779 (2013)

[6] Shin D., and Gupta S.K., A new circuit simplification method for error tolerant applications, DATE 2011, pp.1 (2011)

[7] S. Venkataramani, Sabne A., Kozhikkottu V., Roy K., and Raghunathan A., SALSA: Systematic logic synthesis of approximate circuits, DAC 2012, pp.796 (2012)

[8] Brayton R.K., Logic minimization algorithms for VLSI synthesis, Springer Science & Business Media, 1984