Shanghai Jiao Tong University
University of Michigan - Shanghai Jiao Tong University Joint Institute

Efficient Synthesis Methods for High-Quality Approximate Computing Circuits and Architectures

by

Chang Meng

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy in
Electrical and Computer Engineering at Shanghai Jiao Tong University

Committee in charge (in alphabetical order):
Prof. Zhigang Ji
Prof. Honglan Jiang
Prof. Weikang Qian, Chair
Prof. Paul Weng
Prof. An Zou

Shanghai
May, 2023


20006301

上海交通大学
交大密西根学院

面向高质量近似计算电路和架构的高效综合方法

孟畅

上海交通大学密西根学院博士学位论文
电子科学与技术专业

论文答辩委员会成员（按姓氏排序）：
纪志罡 教授
姜红兰 副教授
钱炜慷 副教授 主席
Paul Weng 副教授
邹桉 助理教授

上海
2023年5月


20006301

Abstract

As modern *very large scale integration* (*VLSI*) designs encompass more and more complexity and the size of transistors shrinks into the nanoscale, it has been increasingly difficult to improve the area, delay, and power of computing systems by conventional design methods. Fortunately, many widely-used applications are error-tolerant, such as data mining, image processing, and machine learning. The property of error tolerance inspires a novel design paradigm called approximate computing, which can further improve the quality by introducing errors into computing systems. With carefully-designed errors, the overall functionality of the application is almost unaffected, while the area, delay, and power can be reduced dramatically. Approximate computing can be applied to all layers of modern computing systems, *i.e.*, the circuit, architecture, and software layers. This dissertation focuses on the first two layers and develops efficient synthesis methods for high-quality approximate circuits and architectures.

At the circuit layer, one popular way to generate approximate circuits is *approximate logic synthesis* (*ALS*), which takes as inputs an exact circuit and user-specified error constraints, and produces an approximate design satisfying the constraints automatically. The performance of an ALS algorithm can be evaluated from two aspects, *i.e.*, *quality and efficiency*. We prefer ALS algorithms that can synthesize *high-quality* (*i.e.*, with small areas, delays, and powers) approximate circuits *efficiently* (*i.e.*, with short runtime).

To improve the quality of ALS algorithms, this dissertation develops area- and delay-driven ALS methods. For one thing, Chapters 2 and 3 focus on reducing the circuit area. Chapter 2 proposes a novel *local approximate change* (*LAC*), which is a local simplification of a sub-circuit, based on *approximate resubstitution*. It can help dramatically reduce the circuit area. Based on the LAC, a novel area-driven ALS flow is developed. The experimental results demonstrate that the proposed flow can reduce $7\% \sim 18\%$ area compared to the state-of-the-art methods. Chapter 3 studies the order of applying LACs, since the order highly affects the circuit quality. Two advanced order search algorithms, beam search and Monte Carlo tree search, are integrated into ALS. The experimental results show that compared to the traditional order based on the greedy strategy, both algorithms can generate better orders to apply LACs, which leads to further improvement of the areas of approximate circuits. For another, Chapter 4 focuses on reducing the circuit delay. It proposes to establish a *critical error graph* consisting of nodes on the critical paths and error information, and applies an optimized set of LACs in the critical error graph to shorten all critical paths so that the circuit delay is reduced. Based on this, a novel delay-driven ALS flow is designed. The experimental results demonstrate that the proposed flow can reduce the circuit delay by 40.7% compared to the state-of-the-art method.

To improve the efficiency of ALS methods, this dissertation tries to accelerate the most time-consuming step of ALS, *error evaluation*. Chapter 5 focuses on one of the widely-used error metrics, the *maximum error*, and proposes an efficient maximum error checking method. It depicts the behavior of the maximum error with *partial Boolean difference* and performs efficient maximum error checking with satisfiability sweeping. Based on this, an efficient ALS flow is designed to synthesize approximate circuits under the maximum error constraint. The experimental results show that the proposed flow is $13\times$ faster than the state-of-the-art ALS method.

At the architecture layer, this dissertation focuses on a popular computation scheme, *i.e.*, approximate computing with memory. It pre-computes a function and stores essential features of the function into a *lookup table* (*LUT*), which is implemented by memory units. Then, we can efficiently compute the function by looking

up entries from the LUT. In this dissertation, Chapter 6 designs a low-power and high-speed approximate LUT architecture. To implement an arbitrary function with the architecture at the cost of the smallest error, efficient synthesis algorithms are proposed using the integer linear programming-based and heuristic-based methods. The experimental results show that the proposed architecture achieves energy and latency savings by 56.5% and 92.4%, respectively, over the state-of-the-art approximate LUT architecture.

To sum up, this dissertation proposes novel techniques that can efficiently synthesize high-quality approximate circuits and architectures. They advance the study of approximate computing and pave a promising way to improve the quality of VLSI designs in the post-Moore era.

摘要

随着当代超大规模集成（very large scale integration, VLSI）电路设计复杂度的提升及晶体管尺寸的缩小，传统设计方法在降低计算系统的面积、延迟和功耗等方面日益受限。 然而，诸如数据挖掘、图像处理和机器学习等常见应用具有容错性。 针对这一特性，研究者们提出了名为“近似计算”的新型设计范式。 通过向计算系统中引入误差， 近似计算能进一步提高芯片质量。 巧妙地引入误差可在几乎不影响应用整体功能的同时， 显著降低芯片的面积、延迟和功耗。 近似计算可用于现代计算系统的所有层次中，即电路层、架构层和软件层。本文主要研究前两层， 提出了面向高质量近似计算电路和架构的高效综合方法。

在电路层， “近似逻辑综合”（approximate logic synthesis, ALS）常用于自动生成近似电路。 给定准确电路以及误差约束， 近似逻辑综合可生成满足约束的近似电路。 我们可以从质量和效率两方面评估近似逻辑综合方法的性能， 期望它高效地（运行时间短）生成高质量（面积、延迟和功耗小）近似电路。

为提高近似逻辑综合方法质量， 本文分别研究了优化面积和优化时延的近似逻辑综合方法。 首先，第2章和第3章关注电路面积优化。 第2章提出了一种基于“近似重替换”的子电路结构化简方式， 这种“局部近似变换”（local approximate change, LAC）方式能够显著减少电路面积。 在此基础上， 第2章设计了一个优化面积的近似逻辑综合流程。 与现有技术相比的实验结果表明，该流程可减少7%到18%的面积。 由于应用局部近似变换的顺序对近似电路的质量影响极大， 第3章围绕此顺序展开研究， 将两种“束搜索”算法和“蒙特卡洛树搜索”算法应用于近似逻辑综合流程。 与传统贪心策略相比的实验结果表明， 这两种顺序搜索算法可优化局部近似变换的应用顺序， 从而降低近似电路的面积。 接下来，第4章重点研究了电路时延的优化， 提出了一种由“关键路径”和误差信息组成的“关键误差图”。 利用关键误差图，我们可以找到一组最优的局部近似变换， 以缩短所有关键路径并降低电路时延。 在此基础上， 第4章设计了一个优化时延的近似逻辑综合流程。 与现有技术相比的实验结果表明，该流程可以减少40.7%的时延。

为提升近似逻辑综合方法的效率， 本文试图加速近似逻辑综合中最耗时的步骤——误差评估。 第5章针对一种常用的误差度量——“最大误差”， 提出了一种快速检查最大误差的方法。 该方法利用“偏布尔差分”的数学工具来分析最大误差， 并利用“可满足性扫描”来实现快速的最大误差检查。 在此基础上，第5章还设计了一个高效的近似逻辑综合流程， 以此快速地生成满足最大误差约束的近似电路。 实验结果表明，该流程比现有技术快13倍。

在架构层， 本文重点研究一种流行的计算方案——基于内存的近似计算。 该方案提前计算好一个函数，并将该函数的特征数据存储在由内存单元实现的“查找表”（lookup table, LUT）中， 从而通过查表实现高效计算。 第6章提出了一种低功耗、高速率的近似查找表架构。 为了尽可能减少用该架构实现函数的误差， 第6章还提出了基于整数线性规划和基于启发式方法的高效综合算法。 与现有技术相比的实验结果表明，该架构可降低56.5%的能耗和92.4%的延迟。

综上所述，本文提出了面向高质量近似电路和近似架构的高效综合技术， 它们推动了近似计算的研究， 并在后摩尔时代为进一步提高芯片质量提供了有效途径。


20006301

Contents

Abstract iii

Nomenclature x

List of Figures xiii

List of Tables xvi

1 Introduction 1
- 1.1 Background of Approximate Computing . . . 1
- 1.2 Approximate Computing Techniques at the Circuit Layer . . . 4
  - 1.2.1 Manual Design . . . 4
  - 1.2.2 Automatic Synthesis . . . 7
- 1.3 Approximate Computing Techniques at the Architecture Layer . . . 10
  - 1.3.1 Approximate Computing with General-Purpose Processors . . . 10
  - 1.3.2 Approximate Computing with Memory . . . 12
- 1.4 Challenges and Motivations . . . 13
  - 1.4.1 Challenges and Motivations on Design Quality . . . 14
  - 1.4.2 Challenges and Motivations on Design Efficiency . . . 15
- 1.5 Overview of the Dissertation Studies . . . 16
- 1.6 Organization of the Dissertation . . . 18

2 ALSRAC: Area-Driven Approximate Logic Synthesis Flow by Resubstitution with Approximate Care Set 19
- 2.1 Motivations and Overview . . . 19
- 2.2 Preliminaries . . . 21
  - 2.2.1 Circuit Terminologies . . . 21
  - 2.2.2 Average Error Metrics . . . 22
- 2.3 Methodology . . . 23
  - 2.3.1 Approximation of Care Set . . . 23
  - 2.3.2 Proposed Local Approximate Change . . . 25
  - 2.3.3 Proposed ALS Flow . . . 29
- 2.4 Experimental Results . . . 30

2.4.1 Experiment Setup . . . 30
2.4.2 Experiments on ASIC Designs . . . 31
2.4.3 Experiments on FPGA Designs . . . 33
2.5 Summary . . . 35

3 Advanced Ordering Search Techniques for Approximate Logic Synthesis 37
3.1 Motivations and Overview . . . 37
3.2 Preliminaries . . . 39
3.3 Methodology . . . 39
3.4 ALS by Beam Search . . . 40
3.5 ALS by Monte Carlo Tree Search . . . 41
3.6 Experimental Results . . . 46
3.6.1 Comparison of Approximate Circuit Quality for Different Methods . . . 47
3.6.2 Comparison of Runtime for Different Methods . . . 49
3.6.3 Quality Configurable ALS Flow with MCTS . . . 49
3.7 Summary . . . 50

4 HEDALS: Highly Efficient Delay-Driven Approximate Logic Synthesis Flow 51
4.1 Motivations and Overview . . . 51
4.2 Preliminaries . . . 52
4.3 Overall Framework of HEDALS . . . 53
4.4 Critical Error Graph . . . 55
4.4.1 LAC Sets for Delay Reduction . . . 55
4.4.2 Promising Subset of All Delay-Reducing LAC Sets and Critical Error Graph . . . 56
4.4.3 Building Critical Error Graph . . . 57
4.5 Obtaining an Optimized LAC Set . . . 58
4.5.1 Maximum Flow-Based Method . . . 58
4.5.2 Priority Cut-Based Method . . . 62
4.6 Experimental Results . . . 67
4.6.1 Experimental Setup . . . 67
4.6.2 Study under the NMED Constraint . . . 68
4.6.3 Study under ER Constraint . . . 74
4.6.4 Study under MHD Constraint . . . 75
4.6.5 Study on Adders and Multipliers . . . 76
4.6.6 Comparison of HEDALS Performance with Different LAC Types and Circuit Representations . . . 78
4.7 Summary . . . 79

5 MECALS: Efficient Maximum Error Checking Technique for Approximate Logic Synthesis 81
5.1 Motivations and Overview . . . 81
5.2 Preliminaries . . . 84
5.2.1 Partial Boolean Difference . . . 84
5.2.2 Maximum Error Metrics . . . 85

5.3 MECALS Methodology . . . 86
5.3.1 A Necessary and Sufficient Condition for a LAC to be Valid . . . 86
5.3.2 Details of MECALS . . . 88
5.3.3 Construction of the PBD Circuit . . . 90
5.4 ALS Flow Based on MECALS . . . 94
5.5 Results . . . 95
5.5.1 Experiment Setup . . . 95
5.5.2 Accuracy-Efficiency of MECALS . . . 95
5.5.3 Comparison of MECALS-Based ALS Flow with MUSCAT . . . 98
5.6 Summary . . . 99

6 DALTA: Decomposition-Based Approximate Lookup Table Architecture 101
6.1 Motivations and Overview . . . 101
6.2 Preliminaries . . . 102
6.3 Key Idea: Approximate Disjoint Decomposition . . . 103
6.4 Decomposition-Based Approximate LUT Architecture . . . 105
6.5 Efficient Synthesis Methods Based on Approximate Decomposition . . . 106
6.5.1 Separate Decomposition of Component Functions . . . 107
6.5.2 Joint Decomposition of Component Functions . . . 109
6.6 Experimental Results . . . 114
6.6.1 Experiment Setup . . . 114
6.6.2 Comparison of Different Decomposition Methods . . . 115
6.6.3 Experiments on Continuous Functions . . . 116
6.6.4 Experiments on Non-Continuous Functions . . . 117
6.6.5 Exploration of Different Structures of DALTA . . . 118
6.7 Summary . . . 119

7 Conclusions and Future Works 121
7.1 Summary of Contributions . . . 121
7.2 Future works . . . 123

Acknowledgements 125

Publication List 127

Bibliography 136


20006301

Nomenclature

AIG	AND-inverter graph
ALS	Approximate logic synthesis
AT	Arrival time
BDD	Binary decision diagram
CEG	Critical error graph
CEN	Critical error network
EDA	Electronic design automation
ER	Error rate
EXDC	External don't-care
FPGA	Field programmable gate arrays
ILP	Integer linear programming
ISOP	Irredundant sum-of-products
LAC	Local approximate change
LSB	Least significant bit
LUT	Lookup table
MaxSE	Maximum square error
MC	Monte Carlo

MCTS Monte Carlo tree search

MED Mean error distance

MEI Minimum error increase

MFFC Maximum fanout-free cone

MHD Mean Hamming distance

MIG Majority-inverter graph

MRED Mean relative error distance

MSB Most significant bit

NMED Normalized mean error distance

NMHD Normalized mean Hamming distance

PBD Partial Boolean difference

PI Primary input

PO Primary output

SAT Satisfiability

STA Static timing analysis

TFI Transitive fanin

TFO Transitive fanout

UCT Upper confidence bound for trees

VLSI Very large scale intergration

WCE Worst-case error

List of Figures

1.1 Sources of error tolerance. 2
1.2 An example of approximating a 2-bit multiplier. This figure is from the reference (Kulkarni et al., 2011). 3
1.3 Approximate logic synthesis (ALS). 8
1.4 A enhanced general-purpose processor that supports both approximate and exact computing. The top part contains precise data paths, and the bottom part contains approximate data paths. This figure is from the reference (Xu et al., 2015). 11
1.5 Design space for traditional and approximate computing systems. 14
1.6 An overview of the dissertation studies. 16

2.1 An example circuit in the AIG representation, where all the edges are non-complemented. 21
2.2 Example circuits. 24

3.1 A greedy ALS method is stuck into a local optimum. A node represents a circuit and an edge represents a LAC. A and E denote the area and error rate of the corresponding circuit, respectively. S is the score of a LAC. 38
3.2 An illustration of beam search with $K = 3$. 40
3.3 The steps of Monte Carlo tree search (MCTS). 43
3.4 Area reduction ratio v.s. error rate for MCTS-ALS. 48
3.5 Circuit quality improves with iteration number for the MCTS-ALS method. The results are obtained on circuit C1908 under ER threshold of 3%. 50

4.1 An example circuit in the AIG representation, where all the edges are non-complemented. The ATs of the nodes are marked in blue. The red nodes are on the critical paths. 53

4.2 An example of shortening all the critical paths in the critical graph by applying delay-reducing LACs to the nodes in a global cut of the critical graph. The left figure is the critical graph of the AIG in Fig. 4.1. 55
4.3 The critical error graph (CEG) of the AIG in Fig. 4.1. The value Δe_i beside the functional node n_i represents the minimum error increase (MEI) of n_i, where the error metric can be any average error metric of interest, such as ER, MED, and MHD. Note that the sink node t does not belong to the CEG; it is only used for obtaining a set of global cuts in Section 4.5.2. 58
4.4 The dual graph built from the CEG shown in Fig. 4.3. 60
4.5 Comparison of estimated error (EER) and actual error (AER). The error metric is NMED. . . . 69
4.6 Comparison between the maximum flow-based and priority cut-based methods on the BACS benchmark suite under an NMED bound of 0.005. 71
4.7 Quality-NMED tradeoff on the BACS benchmark suite. 72
4.8 Quality-ER tradeoff on the ISCAS benchmark suite. 74
4.9 Comparison between the approximate designs synthesized by HEDALS and those from the EvoApproxLib on delay and area ratios versus NMED. 77

5.1 Virtual error circuit to check maximum error. 85
5.2 Virtual error circuits used for maximum error checking. The maximum error bound is B. 87
5.3 Maximum error checking circuit for all LAC g_{ij}'s in the intermediate approximate circuit $\hat{G}$. The LAC g_{ij} replaces the node n_i with a new node $g_{ij} = \phi_{ij}(d_{ij1}, d_{ij2}, \ldots)$. The PBD circuit computes the PBD $\Delta_{n_i} f$. The POs are the validness signals h_{ij}'s, each of which represents the validness of the LAC g_{ij}. The symbols $\oplus$ and $\otimes$ denote XOR and AND, respectively. 89
5.4 Circuit to compute the exact PBD $\Delta_n f$. The symbol $\oplus$ denotes XOR. 89
5.5 Circuit to compute the approximate PBD $\widehat{\Delta}_n f$. 93
5.6 Accuracy-efficiency tradeoff of MECALS. 97

6.1 A 2-dimensional truth table, or Boolean matrix, of a function f. 102
6.2 Boolean matrices of (a) a non-decomposable function and (b) a decomposable function that approximates the function in (a). 104
6.3 Overall decomposition-based approximate LUT architecture. 105
6.4 Approximate single-output LUT. 105
6.5 Comparison of different approximate LUT architectures on continuous functions. 116

6.6 Comparison of DALTA and the rounding architecture on non-continuous functions. 118


20006301

List of Tables

2.1 Node values under all PI patterns for the circuit in Fig. 2.2(a). 24

2.2 A truth table of an approximate function with inputs u and z and output $\hat{v}$ in Example 2.4. . . . 28

2.3 Benchmarks used in experiments. 31

2.4 Comparison of ALSRAC and Su's method under ER constraint. 32

2.5 Comparison of ALSRAC and Su's method under NMED constraint. 33

2.6 Comparison of ALSRAC and Liu's method under ER constraint. 34

2.7 Comparison of ALSRAC and Liu's method under MRED constraint. 34

3.1 Benchmark information 47

3.2 Area reduction ratios (ARRs) of three different ALS methods. 47

3.3 Runtime of the greedy and the MCTS-ALS methods for reaching the same area improvement. T_G and T_M denote the time in seconds of the greedy and the MCTS-ALS methods, respectively. 49

4.1 Benchmark circuit information. 67

4.2 All different optimized LAC sets found by different linear error models in the first 7 iterations of HEDALS. The **bold** entries mean that Eq. (4.3) leads to better LAC sets with smaller AERs than Eq. (4.2). 70

4.3 Comparison between HEDALS and BLASYS under the NMED constraint. The **bold** values mean that HEDALS outperforms BLASYS. 73

4.4 Comparison between HEDALS and BLASYS under the ER constraint. The **bold** values mean that HEDALS is better than BLASYS. 75

4.5 Performance of HEDALS on the large EPFL circuits. The **bold** values mean that HEDALS outperforms BLASYS. Data of BLASYS are from the reference (Ma et al., 2021). N/A means that the corresponding data is not reported in the reference (Ma et al., 2021). 76

4.6 Comparison of HEDALS performance with different LAC types and circuit representations under ER and NMED constraints. "DR" means delay ratio, and "AR" means area ratio. The best choice among the three combinations of LAC type and circuit representation is highlighted in **bold**. . . . 78

5.1 Benchmarks used in experiments 96
5.2 Precision and recall of MECALS over different percentages of nodes using exact PBDs P. 97
5.3 Comparison of MECALS-based flow with MUSCAT. "N/A" means not able to generate approximate circuits in 24 hours. 98

6.1 Comparison of different approximate decomposition methods. 114
6.2 Benchmarks used in the experiments. 115
6.3 Performance of various decomposition methods. 116
6.4 Comparison of NMEDs of DALTA and ApproxLUT with the same LUT size. 118
6.5 Performance of heuristic method for different configurations of DALTA for the $\cos(x)$ function. . 118

Chapter 1

Introduction

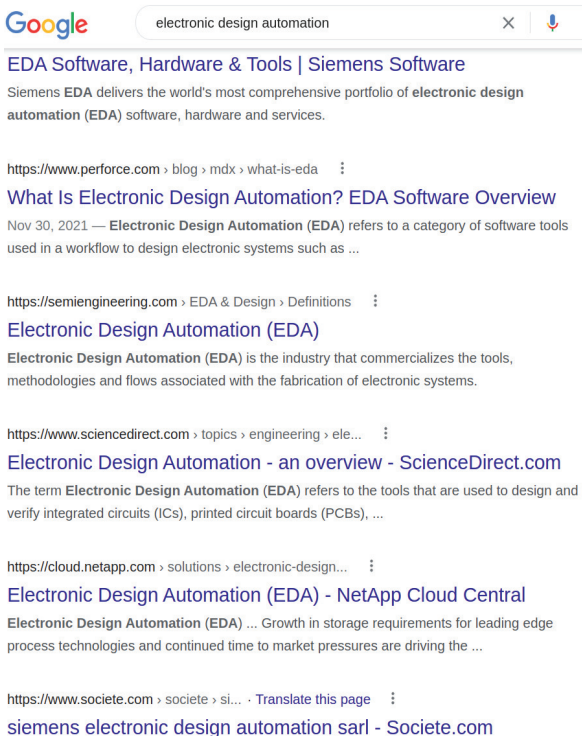
Approximate computing is the central topic of this dissertation. In this beginning chapter, the background of approximate computing will be introduced first. Then, this chapter will describe how to apply approximate computing to different layers of computing systems, and discuss related works of approximation techniques at different layers. After that, challenges in approximate computing will be demonstrated. Finally, an overview of the dissertation studies will be presented, followed by the dissertation organization.

1.1 Background of Approximate Computing

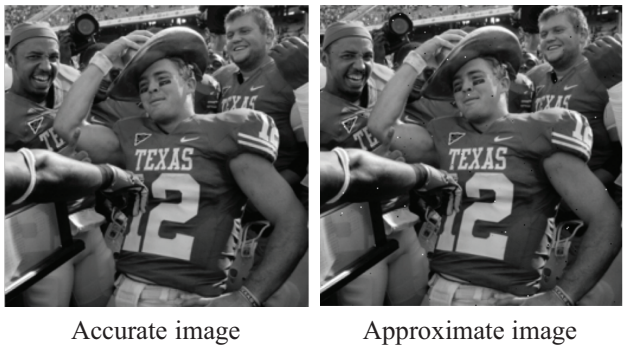
As modern *very large scale integration* (*VLSI*) designs are more and more complex and the transistor size shrinks into the nanoscale, it has been increasingly difficult to improve the area, delay, and power of computing systems by conventional design methods (Waldrop, 2016). Therefore, developing new design paradigms is in high demand to further reduce the hardware cost of computing systems.

Fortunately, many widely-used applications are error-tolerant, such as data mining, image processing, and machine learning. The error tolerance arises from three aspects, as demonstrated in Fig. 1.1. First, many tasks do not have a golden answer. For example, if we search *electronic design automation* (*EDA*) with Google, the top six results are shown in Fig. 1.1(a). It is hard to say which result is the best, and all of them are acceptable for someone wanting to understand EDA briefly. Thus, approximations can be introduced into these applications without golden answers. Second, the perceptual abilities of human beings are limited. For example, Fig. 1.1(b) contains two almost identical images. The left one is exact, while the right one is inexact, with some pepper noises. However, for the inexact image, these noises do not affect the recognition of the sportsman in the image. Third, exact numerical numbers are unnecessary in some scenarios. For instance, Fig. 1.1(c) sketches a

neural network for recognizing handwriting figures. Assume that the network takes a handwriting figure "8" as the input. Then, it computes the similarities between the handwriting figure and each possible figure from 0 to 9. The handwriting figure is most likely to be recognized as 8, and the corresponding similarity is much larger than the similarities of other figures. If we slightly change the simularity values, the recognition result will not be affected.

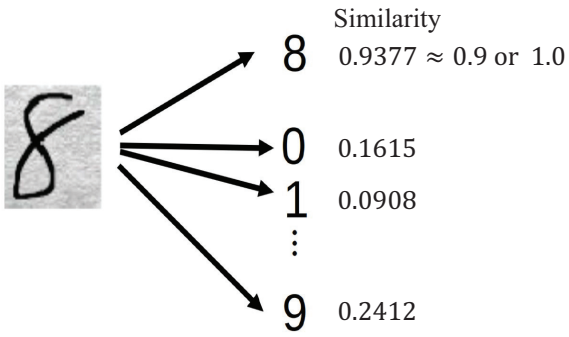


(a) No golden answer.



Accurate image Approximate image

(b) Limitation of human perception.

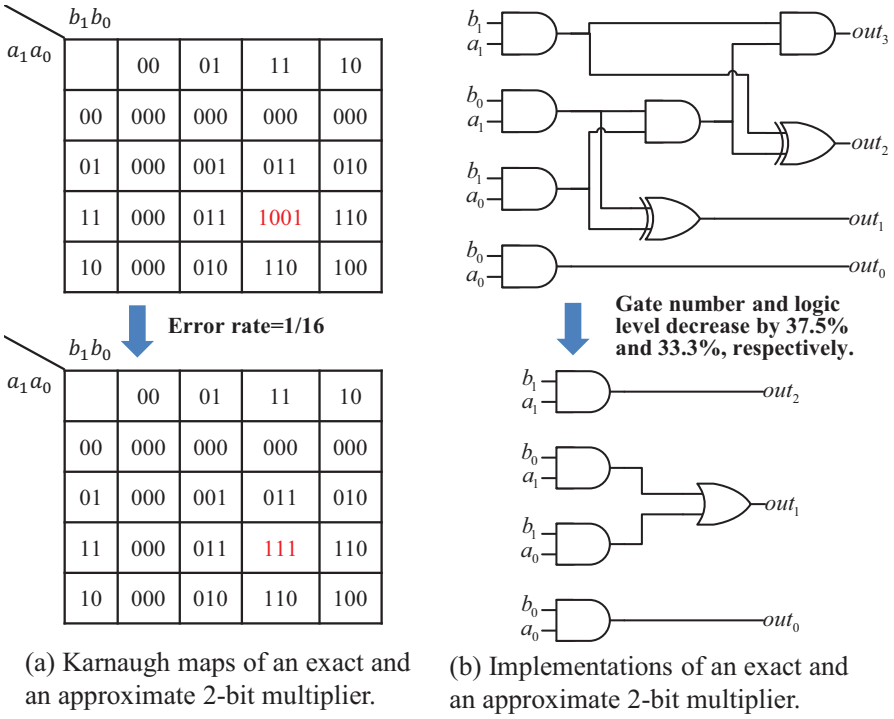


(c) Unnecessity of exact numerical values.

Figure 1.1: Sources of error tolerance.

Targeting at these error-tolerant applications, *approximate computing* is proposed as a novel design paradigm to further improve the performance of computing systems (Han and Orshansky, 2013). Its basic idea is introducing approximation into a target design, such as modifying its function without affecting the overall functionality of the application. With a carefully-designed approximation, the resulting design will have a smaller area, delay, and power than its original version. An example of approximate computing is shown in Fig. 1.2 (Kulkarni et al., 2011). The top of Fig. 1.2(a) shows the Karnaugh map of an exact 2-bit multiplier, and its corresponding exact circuit is shown at the top of Fig. 1.2(b). Suppose that we deliberately modify the output value for the input pattern $a_1a_0b_1b_0 = 1111$ from 1001 to 0111. In that case, the resulting approximate Karnaugh map is shown at the bottom of Fig. 1.2(a), and its corresponding approximate design is shown at the bottom of Fig. 1.2(b).

This approximation introduces an error rate of 1/16. Meanwhile, the corresponding circuit implementation is simplified. As shown in Fig. 1.2(b), the gate number and logic level decrease by 37.5% and 33.3%, respectively, due to the approximation. Therefore, approximate computing can improve the circuit quality at the cost of errors.



(a) Karnaugh maps of an exact and an approximate 2-bit multiplier.

(b) Implementations of an exact and an approximate 2-bit multiplier.

Figure 1.2: An example of approximating a 2-bit multiplier. This figure is from the reference (Kulkarni et al., 2011).

In addition to designing approximate circuits at the circuit layer (*e.g.*, Fig. 1.2), approximate computing can also be applied to other layers of computing systems. As mentioned in the reference (Xu et al., 2015), all layers of modern computing systems, *i.e.*, the circuit, architecture, and software layers, can be approximated. At the circuit layer, we can design high-quality approximate circuits under certain error constraints. At the architecture layer, researchers propose high-quality architectures based on approximate computing. At the software layer, unimportant computations or memory manuplications can be performed approximately to accelerate the computation and reduce the energy consumption. This dissertation focuses on the first two layers, *i.e.*, the circuit and architecture layers, which will be detailed in Sections 1.2 and 1.3, respectively.

1.2 Approximate Computing Techniques at the Circuit Layer

At the circuit layer, there are two categories of methods to design approximate circuits: manual design and automatic synthesis. This section will introduce both of them.

1.2.1 Manual Design

Manual design is widely used to devise approximate arithmetic circuits, such as adders and multipliers. Since arithmetic circuits have well-known structures and are frequently used in computing systems, researchers analyze their structures, utilize the properties of arithmetic operations, and then manually introduce approximation into them. However, manual design requires designers to be experienced and familiar with arithmetic circuits. Designers may consume a lot of time to design new approximate arithmetic circuits. Given that adders and multipliers are the most important arithmetic units, how to design approximate adders and multipliers manually will be introduced as follows.

Manual Design of Approximate Adders

Some researchers study how to design high-quality approximate adders manually. Zhu et al. (2009) proposed a low-power and high-speed approximate adder called *ETAII*. Unlike traditional adders, ETAII splits the entire carry chain into several short paths, each of which belongs to a sub-adder. Then, carry signals belonging to different short paths in different sub-adders can be propagated simultaneously, hence improving the speed. As the carry chain are simplified, the area overhead and power consumption of the adder also decrease. Zhu et al. (2009) also proposed another approximate adder. It segments an exact adder into two parts: an accurate part including *most significant bits* (*MSBs*), and an inaccurate part including *least significant bits* (*LSBs*). The length of the two parts can be unequal. The accurate part performs conventional carry propagation without inducing errors to avoid inducing significant errors on MSBs, while the inaccurate part performs inexact addition by ignoring the carry signals to reduce the hardware cost. Kahng and Kang (2012) proposed an *accuracy-configurable approximate adder* (*ACA*) whose accuracy can be configured during runtime. It consists of an approximate adder and an *error detection and correction* (*EDC*) unit. By configuring the EDC unit at runtime, the computing accuracy can be tuned dynamically. Moreover, ACA can achieve significant improvement on delay and power. Gupta et al. (2012) proposed to introduce approximation to the transistor network of a 1-bit *full adder* (*FA*), which reduces the number of transistors at the cost of accuracy reduction. Based on the idea, they designed various approximate FA cells with simplified transistor structures. Then, they utilized the approximate FA cells to construct larger approximate adders. The proposed approximate adders have fewer transistors than

the exact ones, and hence, they also have smaller areas. Besides, since the parasitic capacitance is directly proportional to silicon area, their approximate adders benefit from a dynamic power reduction. Ye et al. (2013) proposed a reconfiguration-oriented approach of designing approximate circuits and designed a reconfigurable approximate adder. To develop high-quality approximate circuits, they utilized a *quality-effort* curve, which depicts the relationship between the circuit quality (*e.g.*, area, delay, and power) and the computational effort (*e.g.*, energy consumption). With the guidance of the curve, they designed an accuracy *gracefully degrading adder* (*GDA*), which features low latency and high accuracy. Shafique et al. (2015) proposed a novel way of splitting an exact adder into sub-adders. Different from ETAII (Zhu et al., 2009), which also performs splitting on adders, their proposed design allows successive sub-adders to share common input bits. The sharing structure improves the accuracy without sacrificing delay. However, it also causes an increase in the area overhead and power consumption. Hu and Qian (2015) exploited the *generate signals* for carry speculation, and designed a low-power and high-speed approximate adder. The maximum relative error of their adder is controlled by an error reduction unit, and the sign bit is always correct due to the sign correction unit. Camus et al. (2018) proposed a novel technique to optimize approximate circuits by fabricating unactivated critical paths. Based on this, they proposed a new approximate adder called *carry cut-back adder* (*CCBA*). High-significance portions of CCBA are approximated by cutting the carry chains at LSBs, which reduces the circuit delay and power significantly.

Manual Design of Approximate Multipliers

Some researchers study how to design high-quality approximate multipliers manually. Kyaw et al. (2010) proposed to approximate a multiplier by dividing it into two parts, the multiplication and non-multiplication parts. The multiplication part is applied to some MSBs of the multiplier, which performs accurate multiplication. The non-multiplication part, which is implemented by skipping the computation of partial products, is applied to the LSBs. Hence, the hardware cost of the non-multiplication part is reduced at the cost of some errors. Kulkarni et al. (2011) proposed a 2×2 underdesigned multiplier block. Then, they utilized the block to construct larger low-power approximate multipliers. The constructive way of designing approximate multipliers also allows tuning the multiplier accuracy flexibly. Lin and Lin (2013) proposed to build a basic approximate 4×4 Wallace-tree multiplier with low power consumption. Based on it, they further constructed larger approximate Wallace-tree multipliers. They also used an EDC unit to reduce the errors. Liu et al. (2014b) designed a new approximate adder, in which the carry propagation is only limited to the nearest neighbors. With the newly-designed adder, they proposed a novel approximate multiplier that performs efficient partial product accumulation. Moreover,

they applied an error recovery technique that can configure the multiplier into different accuracy levels. Jiang et al. (2015) proposed a novel approximate technique to design high-speed and power-efficient approximate radix-8 booth multipliers. They first devised a low-overhead approximate 2-bit adder. Then, they applied the 2-bit adder to design an approximate recording adder, which is an essential component to generate the triple multiplicand in the radix-8 multiplication. Based on the recording adder, they further designed two structures of approximate radix-8 booth multipliers with excellent tradeoffs between circuit quality and accuracy. Rehman et al. (2016) presented an architecture exploration method for approximate multipliers. Three crucial selections are considered in their method: different types of approximate multiply units, different types of adder used to accumulate the partial products, and the selection of approximated bits. With the proposed method, they created a library of high-quality approximate multipliers. Liu et al. (2017) designed an approximate radix-4 Booth multiplier by approximately simplifying the Booth encoding algorithms. Their approximate multiplier features a regular partial product array using an approximate Wallace tree. They also used an approximation factor to depict the error behavior of the proposed multiplier. Imani et al. (2017) proposed a low-power approximate floating-point multiplier called *CFPU*, which can dynamically detect the inputs causing large errors. For inputs causing large errors, CFPU performs exact multiplication. For other inputs not inducing large errors, CFPU replaces the most-costly step of the multiplication by a lower-power alternative to save power. Saadat et al. (2018) devised a *minimally biased approximate integer multiplier* (*MBM*), whose error can be configured. It combines a novel error-reduction mechanism with an approximate logarithmic-based integer multiplier. Moreover, they further simplified the MBM structure and employed the simplified MBM to design approximate float-point multipliers with small areas and powers. Esposito et al. (2018) proposed novel approximate compressors. Then, they developed an algorithm to design high-quality approximate multipliers with the compressors. The resulting multipliers have lower power or faster speed. Vahdat et al. (2019) proposed a *truncation- and rounding-based scalable approximate multiplier* (*TOSAM*). TOSAM truncates each input operand according to its position of the leading one-bit. Then, the number of partial products can be reduced. TOSAM also performs light-weighted multiplication using simple operations such as shifting. Furthermore, the accuracy of TOSAM is improved by a smart rounding technique. Ansari et al. (2019) proposed a high-quality approximate logarithmic multiplier. They first approximated the logarithmic operation by rounding to its nearest power of two. Based on it, they further designed two enhanced logarithmic multipliers, which use exact and approximate adders, respectively. Van Toan and Lee (2020) focused on designing approximate multipliers with *field programmable gate arrays* (*FPGAs*). They first designed approximate logic compressors with different errors and then used them to construct approximate multipliers on FPGAs.

1.2.2 Automatic Synthesis

Although we can manually design high-quality approximate adders and multipliers, applying manual design to arbitrary circuits is impossible. The reason is that we often have little prior knowledge of an arbitrary circuit, such as a regular circuit structure and a well-known data flow of computation, which brings difficulties to its manual design. To address the issue, we can design an algorithm to automatically synthesize an approximate version of an arbitrary circuit. One topic of this dissertation is the automatic synthesis of approximate circuits, which can be divided into *approximate high-level synthesis* (*AHLS*) and *approximate logic synthesis* (*ALS*). Recent progress on AHLS and ALS will be introduced as follows.

Approximate High-level Synthesis (AHLS)

AHLS approaches target to integrate inexact operators as building blocks to implement circuits described in high-level languages, such as the behavioral Verilog or C language. A few representative AHLS methods are discussed here. Nepal et al. (2014) proposed an AHLS framework called *ABACUS*, which is one of the first AHLS techniques. ABACUS takes a behavioral description of a circuit as its input. Based on the behavioral description, it creates a corresponding *abstract synthesis tree* (*AST*). Then, ABACUS applies various operators to the AST and finds the best approximate circuit with an iterative method. Finally, ABACUS produces an actual implementation of the approximate circuit, such as a gate-netlist. Li et al. (2015) proposed novel techniques that use basic arithmetic units to construct large computing systems. They designed approaches to implement approximate scheduling, functional unit allocation, and binding algorithms. They also proposed credible but low-cost error models for evaluating the system accuracy. Based on the models, they further developed a sequential heuristic and an integer linear programming-based approach for joint precision optimization and approximation-aware high-level synthesis. Lee and Gerstlauer (2017) tried to reduce the number of clock cycles to finish an operation in the AHLS process. They optimized the number of clock cycles with the operation elimination technique, and selectively approximated some loop iterations based on their effects on the accuracy.

Approximate Logic Synthesis (ALS)

ALS techniques operate on a gate-netlist or a Boolean representation of the circuit. As shown in Fig. 1.3, ALS takes an exact circuit (represented in a gate-netlist or a Boolean representation) and user-specified error constraints as inputs, and returns a high-quality approximate circuit satisfying the constraints. One of the topics this dissertation focuses on is ALS. In what follows, we review how to introduce approximation into a circuit to simplify the circuit, meanwhile improving the circuit quality.

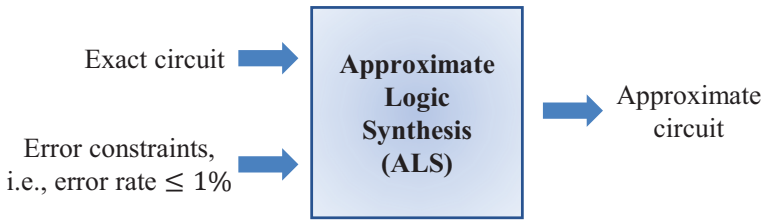


Figure 1.3: Approximate logic synthesis (ALS).

Most ALS methods approximate a circuit by iteratively applying *local approximate changes* (*LACs*), which are local modifications on sub-circuits of the circuit. For example, Shin and Gupta (2011) proposed a LAC that replaces a node in a circuit by a constant 0 or 1. After the substitution, the constant signal can be propagated in the circuit, resulting in additional area reduction, potential delay reduction, and potential power reduction. Venkataramani et al. (2013) also proposed a LAC called SASIMI, which substitutes a node u in the circuit with another node v or its negation. If the function of u is similar to that of v or v's negation, then such a substitution induces a small error. After the substitution, the *maximum fanout-free cone* (*MFFC*, defined in the reference (Cong et al., 1999)) of node u can be removed, reducing the area and possibly the delay and power. Wu and Qian (2016) proposed an approximate node simplification technique. Its LAC is deleting some literals from the Boolean expression of a node in the circuit, which can lead to area reduction after technology mapping. They also developed an efficient dynamic programming-based ALS flow to select multiple LACs in each iteration, which accelerates the ALS flow. Chandrasekharan et al. (2016) proposed an ALS method based on rewriting the *AND-inverter graph* (*AIG*). To simplify the circuit, they proposed to select a cut of a certain node in the AIG and rewrite the cut with a constant 0. This technique is good at reducing the circuit delay. Besides, it supports both average and maximum error metrics. Soeken et al. (2016) focused on circuits represented in *binary decision diagrams* (*BDDs*) and proposed two kinds of LACs to approximate BDDs. The first LAC is replacing a BDD node with one of its children (*i.e.*, a *cofactor* of the BDD node). The second LAC is rounding, by which a child of a BDD node is replaced by a constant 0 or 1. Based on the two LACs, they developed a greedy algorithm to minimize the size of BDDs. Froehlich et al. (2017) pointed out that the approach proposed in the reference (Soeken et al., 2016) does not produce the best BDD with the smallest size under a given error constraint. It is better to find the best BDD with the minimal size for a given error bound. To do this, they constructed a generic BDD, which represents all possible functions that satisfy the error constraint. Then, from the generic BDD, the best BDD structure satisfying the error constraint can be obtained. However, their method only works on simple functions, since the structure of the BDD representing all possible functions is too complex. Liu and Zhang (2017) proposed a stochastic ALS framework including various LACs, *i.e.*, removing a gate, flipping a local output, and adding a gate. For one thing, removing a gate

can reduce the circuit area and possibly also the circuit delay and power. For another, flipping a local output and adding a gate can change the circuit structure and provide potential opportunities to reduce the hardware cost. This framework features stochastic optimization, which may lead to better approximate circuits than conventional ALS flows. In addition, the framework can guarantee that the resulting approximate circuits have a statistically certified error. Schlachter et al. (2017) proposed an ALS flow called *gate-level pruning* (*GLP*), which works on gate netlists. Its LAC is pruning a gate from a circuit, *i.e.*, replacing a gate with a constant 0 or 1. Different from the naive constant substitution proposed in the reference (Shin and Gupta, 2011), in the GLP flow, gates are pruned according to two criteria. The first criterion is the node significance, which represents the impact of the node on the final output. The second criterion is the node activity or toggle count of the node. GLP applies an iterative strategy to prune gates based on the two criteria. Each iteration prunes the gate with the lowest significance, the lowest activity, or the lowest significance-activity product. Scarabottolo et al. (2018) proposed an ALS flow called *circuit carving* (*CC*). As opposed to the reference (Schlachter et al., 2017), CC does not apply LACs iteratively. Instead, CC detects the largest sub-circuit of an exact circuit that can be removed without violating the error constraint. Then, CC discards the detected sub-circuit to reduce the hardware cost of the circuit. To detect the largest sub-circuit to be removed, they devised an algorithm based on binary tree exploration. Ma et al. (2021) proposed an ALS method called *BLASYS*. By Boolean matrix factorization, its LAC is approximately decomposing a sub-circuit into two units, *i.e.*, a compressor unit and a decompressor unit. The area, delay, and power of the sub-circuit can be reduced if the decomposition is done properly. Witschen et al. (2022) proposed an ALS method called *MUSCAT*, which formulates the ALS problem as a minimal unsatisfiable subset problem for solving. Then, MUSCAT applies formal verification engines to identify the minimal unsatisfiable subsets. Its LAC is also replacing a node in the circuit by a constant 0 or 1 (Shin and Gupta, 2011).

There are also several ALS methods that do not depend on an iterative application of LACs. For example, Venkataramani et al. (2012) proposed an ALS method called *SALSA*. It identifies *don't-cares* in a circuit based on the error constraint and then converts the ALS problem into a traditional logic synthesis problem with don't-cares. However, SALSA is not scalable to large circuits, since it represents the don't-cares in terms of *primary inputs* (*PIs*). The total number of don't-cares may grow exponentially with the PI number, making it intractable to represent all don't-cares and formulate the resulting logic synthesis problem. Ranjan et al. (2014) extended SALSA and proposed a new ALS method called *ASLAN* to synthesize sequential circuits. ASLAN measures how the approximation of combinational blocks affects the final outputs with a circuit block exploration approach. Then, ASLAN applies a gradient-descent method to find high-quality approximate circuits. Miao

et al. (2013) proposed an ALS technique under general error magnitude and frequency constraints. They first considered the ALS problem that is only constrained by the error magnitude and converted it into a well-studied Boolean relation minimization problem. Then, they solve the ALS problem constrained by the error magnitude constraint and the frequency constraint simultaneously with the help of a heuristic method. In this way, they can obtain a solution that satisfies both the error magnitude and frequency constraints. Vasicek and Sekanina (2014) proposed an evolution-based ALS framework. Their design process is based on *Cartesian genetic programming* (*CGP*), which encodes circuits into chromosomes, evolves new approximate designs by mutation, and selects promising approximate designs of high accuracy with a fitness function. Using the CGP-based ALS flow, they created a library of approximate adders and multipliers called *EvoApproxLib*.

1.3 Approximate Computing Techniques at the Architecture Layer

At the architecture layer, the most important components that constitute a computer are the processor and memory (Xu et al., 2015). Approximate computing can be applied to both of them to reduce the area, delay, and power by sacrificing accuracy. Although this dissertation mainly focuses on approximate computing with memory, a complete literature review at the architecture layer will be provided.

1.3.1 Approximate Computing with General-Purpose Processors

As discussed in the reference (Xu et al., 2015), if we want to perform an approximate version of a generic program with a general-purpose processor, we should first analyze the program, and determine which instructions in the program can be approximated and which ones cannot tolerate any error. Then, we can use the processor to approximately execute the error-tolerant instructions and exactly executes the error-free instructions. To do this, we need to improve the traditional general-purpose processor so that it can perform both approximate computing and exact computing. Specifically, the new architecture of the improved processor usually has two kinds of data paths, precise and approximate data paths. As shown in Fig. 1.4, the top part contains precise data paths, and the bottom part contains approximate data paths. There are many methods to implement the approximate data paths, such as aggressive voltage scaling (Esmaeilzadeh et al., 2012) or approximately modifying the corresponding circuit structure of the data path.

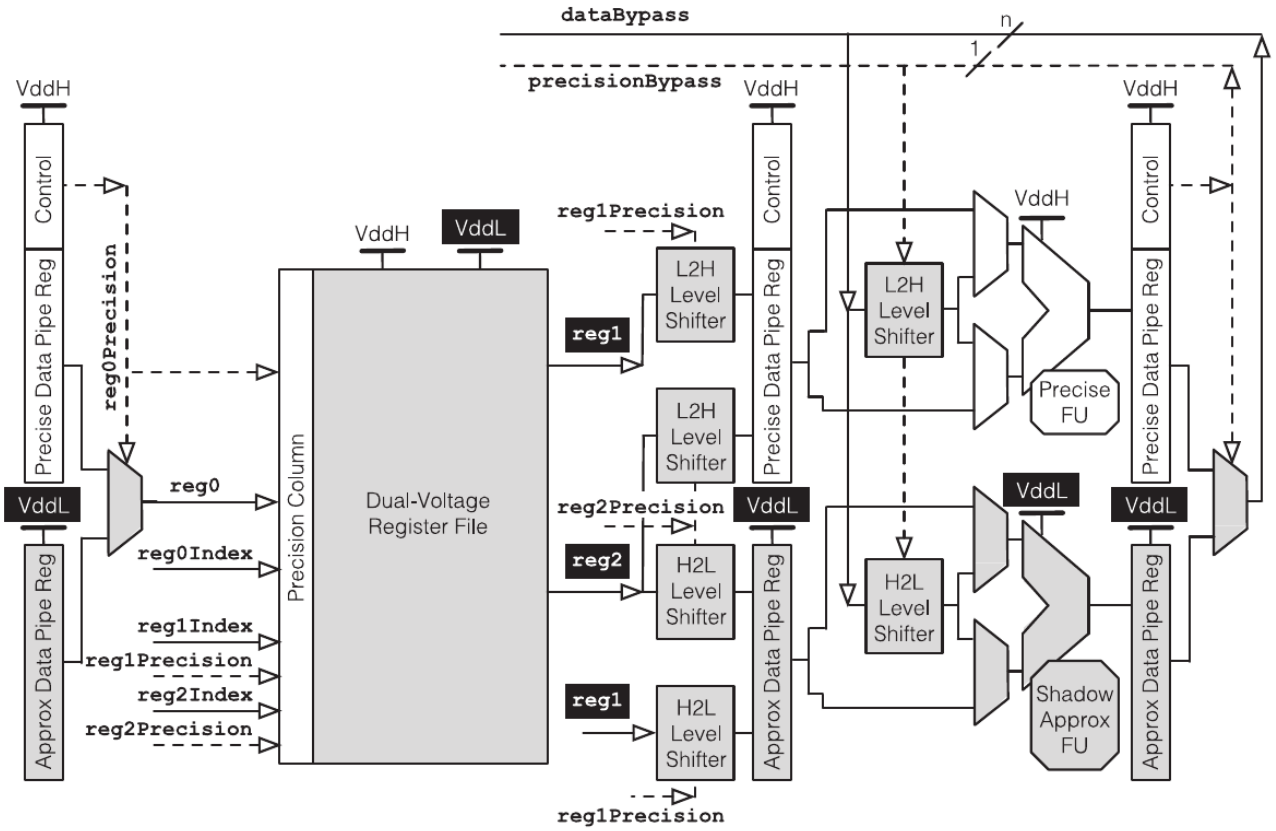


Figure 1.4: A enhanced general-purpose processor that supports both approximate and exact computing. The top part contains precise data paths, and the bottom part contains approximate data paths. This figure is from the reference (Xu et al., 2015).

1.3.2 Approximate Computing with Memory

There are two kinds of methods to perform approximate computing with memory. For one thing, we can directly design approximate memory devices and perform approximate computing with them. For another, we can design architectures that perform approximate computing with exact memory devices.

Approximate Computing with Approximate Memory Devices

Many researchers focus on designing approximate memory devices. *Approximate memory* is a special type of memory design. It introduces some errors when storing data or retrieving data so that the hardware cost of the memory is dramatically reduced. For example, Shoushtari et al. (2015) worked on approximate *static random-access memory* (*SRAM*). They summarized the challenges that will be faced when applying approximate SRAM to error-tolerant applications. To address the challenges, they proposed to utilize the relaxed cache exemplar technique, which can dramatically reduce the energy consumption of the SRAM cache at the cost of small errors. Cho et al. (2014) worked on approximate *dynamic random-access memory* (*DRAM*). They devised *tiered-reliability memory* (*TRM*) to reduce the refresh power of embedded DRAM-based frame buffers. Specifically, the frame buffer is divided into several parts with different refresh rates. For a given data, they stored the MSBs into the more reliable parts of TRM with higher refreshing rates, and saved the LSBs into the less reliable parts with lower refreshing rates. Thus, the overall refreshing power can be dramatically saved at the cost of small errors on LSBs. Liu et al. (2011) proposed another technique, called Flikker, to reduce the refresh power of DRAM. Flikker is an application-level technique, which allows users to point out which data is important and which data is unimportant. Then, Flikker allocates data in different parts of memory with different accuracy. The memory containing important data is refreshed regularly so that no error is induced, while the unimportant data is refreshed at lower rates. Thus, the overall power consumption is reduced at the cost of errors. Ranjan et al. (2015) focused on *spin-transfer torque magnetic random-access memory* (*STT-MRAM*) and explored the tradeoff between hardware cost and accuracy in STT-MRAM. They deliberately introduced a small error when reading data from STT-SRAM or writing data into STT-SRAM, which reduces the power consumption. Sampson et al. (2014) worked on *non-volatile memory* (*NVM*) and proposed methods to improve the quality of NVM, such as performance, lifetime, and density. Their first method reduces the number of pulses when writing the NVM. The second method mitigates wear-out failures and enlarges the NVM lifetime by storing the approximate data with the blocks that have already exhausted the error correction resources. Fang et al. (2012) studied *phase change memory* (*PCM*) and proposed a *SoftPCM* technique to reduce power consumption and extend endurance. SoftPCM analyzes the data from the video application and extracts the error tolerance

characteristic. Based on this, SoftPCM deliberately degrades the accuracy of write operations to save power and improve lifetime.

Approximate Computing with Traditional Memory Devices

Many researchers develop approximate computing architectures with traditional memory devices, which is also one of the topics this dissertation studies. One of the popular methods is by approximate *lookup tables* ($LUTs$). Some early works divide the function's input x into segments and split the original LUT into multiple LUTs. Schulte and Stine (1997) proposed a bipartite LUT to approximate functions based on first-order Taylor approximation. It divides the input x into three segments and uses them to index the two LUTs. The lookup results of all the segments are accumulated to approximate the original value $f(x)$. They further extended the bipartite LUT to the multipartite LUT, as discussed in the reference (Stine and Schulte, 1999). Later, researchers further proposed other multipartite LUT architectures (Muller, 1999; De Dinechin and Tisserand, 2005; Hsiao et al., 2016). However, these architectures all rely on the first-order Taylor approximation of the original functions, and hence they do not support non-continuous functions. With the advent of low-cost associative memristive memories, researchers proposed to only store part of the input-output patterns of the original function, and search the LUT by approximate input pattern matching (Rahimi et al., 2015; Imani et al., 2016). Typical pattern-matching measures include the Hamming distance (Rahimi et al., 2015) and the binary encoding distance (Imani et al., 2016). If an input pattern $\hat{x}$ in the LUT is close enough to a query x, it is treated as a successful pattern matching. Then, the corresponding output pattern $f(\hat{x})$ is returned as an approximation to $f(x)$. Otherwise, $f(x)$ is computed by conventional methods. Additional hardware for accurate computing is required when pattern matching fails. Recently, Tian et al. (2017) proposed an approximate LUT architecture called $ApproxLUT$, which applies Taylor approximation to the pattern matching-based method. ApproxLUT stores the input-output patterns $(a, f(a))$, as well as the derivative $f'(a)$. When an input x comes, it matches the input x with the nearest input pattern a and then computes the approximate function as $f(a)+f'(a)(x-a)$. Unfortunately, as it is also based on Taylor approximation, it can only deal with continuous functions. Furthermore, it requires additional arithmetic circuits to obtain the final result.

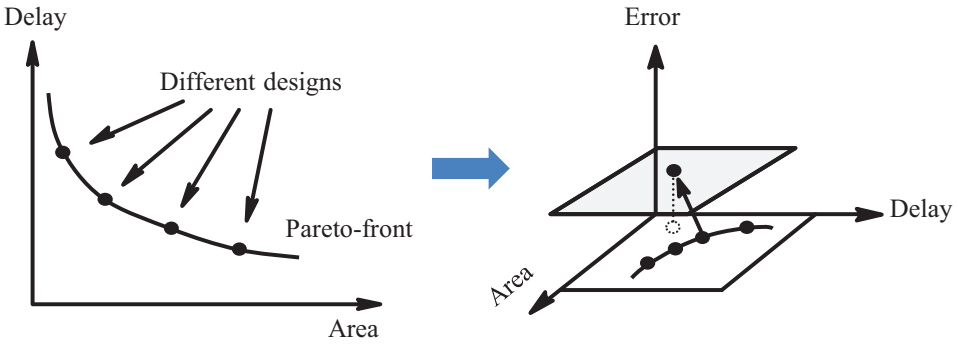
1.4 Challenges and Motivations

As highlighted in the reference (Mittal, 2016), approximate computing encounters many challenges. Addressing these challenges is crucial for making approximate computing practically useful. Major challenges in approximate computing pertain to two aspects: *design quality* and *design efficiency*. The challenges and motivations

of the two aspects will be detailed as follows.

1.4.1 Challenges and Motivations on Design Quality

The first challenge of approximate computing pertains to design quality. As illustrated in Fig. 1.5, unlike traditional computing systems designed within a two-dimensional space, approximate computing systems incorporate error as an additional dimension. This results in a more complex three-dimensional space, complicating the identification of high-quality designs at both the circuit and architecture layers.



(a) 2-dimensional design space for traditional computing systems.

(b) 3-dimensional design space for approximate computing systems.

Figure 1.5: Design space for traditional and approximate computing systems.

At the circuit layer, determinging an optimal approximate circuit with the highest quality, such as the smallest area or delay, is intractable. Although numerous existing methods concentrate on enhancing approximate circuit quality, considerable room for improvement remains. First, numerous studies propose circuit area reduction through area-reducing LACs (Shin and Gupta, 2011; Venkataramani et al., 2013; Wu and Qian, 2016; Yao et al., 2017; Liu et al., 2017). However, LACs utilized in these works often induce substantial errors. For instance, a LAC replacing a node by a constant 0 or 1 (Shin and Gupta, 2011) is excessively crude, potentially causing significant errors. Consequently, fine-grained LACs capable of simplifying circuits at the cost of small errors are necessary, motivating Chapter 2. Second, most existing ALS methods simplify the circuit iteratively (Shin and Gupta, 2011; Venkataramani et al., 2013; Wu and Qian, 2016; Chandrasekharan et al., 2016; Yao et al., 2017; Schlachter et al., 2017; Ma et al., 2021). Each iteration selects the best LAC (*e.g.*, with the smallest error) to simplify the circuit. This greedy-based strategy of applying LACs is often trapped in a local minimum, unable to produce the highest-quality approximate circuit. Notably, different LAC application orders yield distinct approximate circuits with varying quality. Investigating ALS ordering techniques, which can identify optimal LAC application orders to improve circuit quality, is necessary, motivating Chapter 3.

Third, limited research directly studies the delay reduction of approximate circuits. Although the area-driven ALS methods can reduce circuit delay as a byproduct, the potential of ALS for optimizing delay is not fully explored. Therefore, a specific delay-driven ALS framework is highly desirable, motivating Chapter 4.

At the architecture layer, designing approximate computing architectures requires balancing quality and accuracy. Quality refers to the overall performance of the architecture, such as area, delay, and power, while accuracy refers to the precision of computations executed by the architecture. Although the complex three-dimensional space complicates identifying the ideal quality-accuracy tradeoff, manual design of high-quality approximate architectures based on experience remains possible. The difficulty of design such architectures requires urgent attention, serving as the starting point for Chapter 6. The design objective is to maximize quality while minimizing accuracy loss introduced by approximation techniques.

1.4.2 Challenges and Motivations on Design Efficiency

The second challenge of approximate computing concerns design efficiency. Approximate computing confronts a larger design space compared to conventional computing, making it difficult to explore the vast design space and identify high-quality approximate designs quickly.

At the circuit layer, current ALS methods can only handle circuits comprising thousands of gates within several days, as reported in the reference (Su et al., 2022). However, industrial scenarios demand addressing substantially larger circuits, such as a *central processing unit* (CPU) containing millons of gates. Consequently, enhancing ALS efficiency is crucial for practicality. As ALS methods expend most of their time on error evaluation (Su et al., 2022), acceelrating the error evaluation step offers a promising approach for accelerating ALS. Among various error metrics, the maximum error is extensively utilized with numerous works addressing maximum error evaluation (Scarabottolo et al., 2018; Witschen et al., 2022; Shin and Gupta, 2011; Chandrasekharan et al., 2016). Nevertheless, existing methods exhibit limitations. For example, works in the references (Schlachter et al., 2017; Scarabottolo et al., 2021) are restricted to a simple LAC replacing a signal by a constant 0 or 1 (Shin and Gupta, 2011). They do not support more complex LACs capable of yielding high-quality approximate circuits. Therefore, refining maximum error evaluation methods is essential for applying more complex LACs to enhance circuit quality, motivating Chapter 5.

At the architecture layer, assume that a high-quality approximate computing architecture has been designed, and an error-tolerant application will be deployed on this architecture. The deployment requires a synthesis method that transforms the corresponding function of the application into the hardware implementation (*e.g.*, hardware configurations, stored data, etc.) within the architecture. To facilitate convenient deployment and

make the architecture practically useful, developing efficient synthesis algorithms is important, motivating the study of synthesis algorithms in Chapter 6.

1.5 Overview of the Dissertation Studies

To address the challenges discussed in Section 1.4, this dissertation proposes solutions at the circuit and architecture layers. Fig. 1.6 provides an overview of the dissertation studies. Chapters 2–5 focus on the circuit layer, studying ALS techniques, while Chapter 6 centers on the architecture layer, proposing a high-quality approximate computing architecture and its supporting synthesis methods.

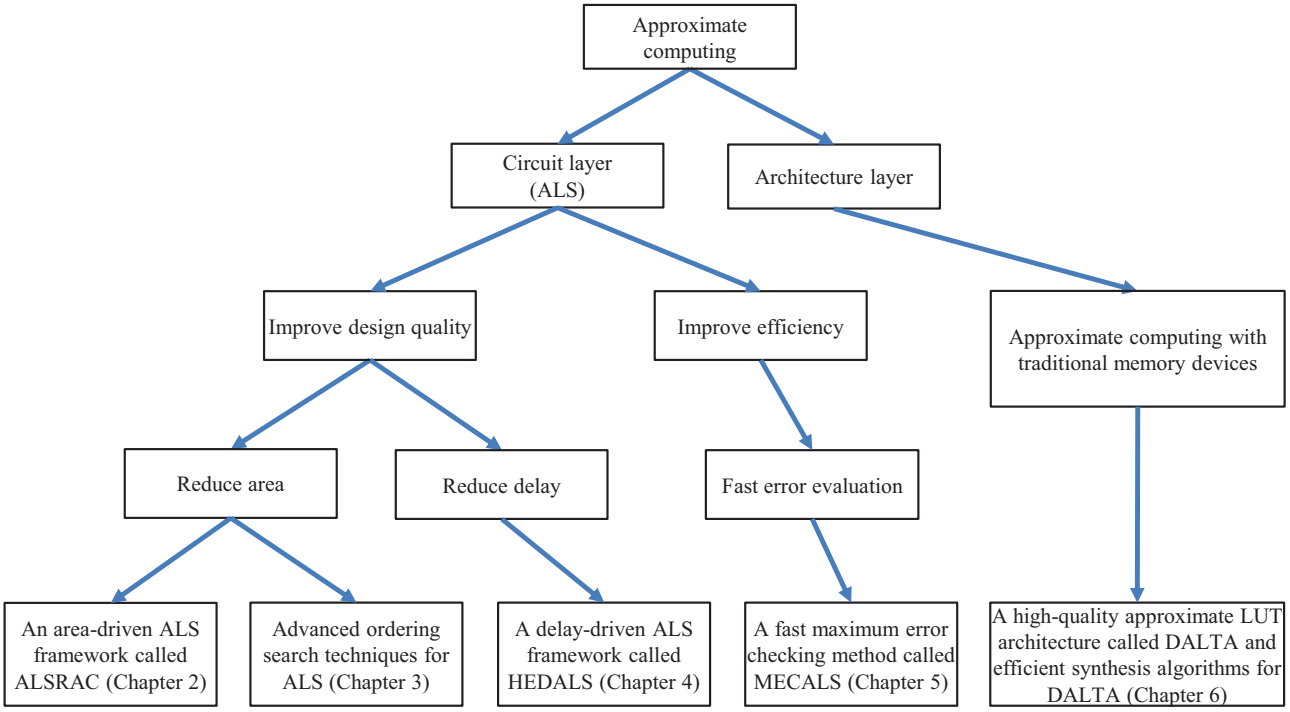


Figure 1.6: An overview of the dissertation studies.

At the circuit layer, ALS algorithms are studied. For one thing, this dissertation studies how to improve the quality of approximate circuits, including approaches of reducing the area and delay. For another, acceleration techniques in ALS are studied.

To reduce the circuit area, Chapter 2 proposes a novel LAC for circuit simplification. Unlike previous LACs that modify the circuit roughly, the proposed LAC utilizes *approximate resubstitution* to approximate the local structure of the circuit in a fine-grained way. Thus, the error introduced by the proposed LAC is much smaller. Based on the proposed LAC, a novel area-driven ALS flow called ALSRAC is developed, which can generate

high-quality approximate circuits with dramatically reduced areas.

To further reduce the circuit area, Chapter 3 studies how to generate a good order of applying LACs, since the order highly affects the quality of final approximate circuits. Unlike traditional greedy-based ordering, Chapter 3 applies two advanced order search techniques, beam search and Monte Carlo tree search, in ALS. The resulting ALS flow enhanced by the two techniques can achieve better LAC orderings, which lead to high-quality approximate circuits with small areas.

To reduce the circuit delay, Chapter 4 works on the critical paths that directly determine the delay. Note that there may exist multiple critical paths in a circuit. Reducing the circuit delay requires shortening all critical paths. To do this, it is proposed to establish a *critical error graph* (*CEG*) containing the critical paths and the error information, and apply an optimized set of LACs on the CEG to shorten all the critical paths simultaneously. Furthermore, a novel delay-driven ALS flow called HEDALS is developed, which can generate high-quality approximate circuits with dramatically reduced delays.

To improve the efficiency of ALS methods, Chapter 5 accelerates the most time-consuming step of ALS, *i.e.*, error evaluation. Among different error metrics, the maximum error is essential and widely used to measure the accuracy of arithmetic circuits. Thus, how to efficiently evaluate maximum errors is studied in Chapter 5. Specifically, the behavior of the maximum error metric is analyzed and modeled, and an important theoretical foundation for checking the maximum error is proposed. Moreover, an efficient maximum error checking method called MECALS is developed. Based on MECALS, an efficient ALS flow is built under the maximum error constraint, which can efficiently check the maximum errors of LACs and generate approximate circuits in a short time.

At the architecture layer, Chapter 6 focuses on approximate computing with traditional memory devices. Specifically, it designs a high-quality approximate computing LUT architecture called DALTA. Unlike previous methods that only support continuous functions (Schulte and Stine, 1997; Muller, 1999; De Dinechin and Tisserand, 2005; Hsiao et al., 2016; Tian et al., 2017), DALTA can implement arbitrary functions regardless of their continuity with high accuracy. It is based on approximate function decomposition and consists of two-level approximate LUTs. Compared to state-of-the-art methods, DALTA consumes less energy and features low latency. Furthermore, to implement an arbitrary function with the DALTA architecture with high accuracy, fast synthesis methods based on integer linear programming and a heuristic method are proposed. These methods can efficiently find an optimal approximate decomposition with the smallest error.

1.6 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 will show the area-driven ALS framework called ALSRAC. Chapter 3 will describe advanced ordering search techniques for ALS that can help reduce circuit area. Chapter 4 will present the delay-driven ALS framework called HEDALS. Chapter 5 will introduce an efficient maximum error checking method called MECALS, which can dramatically accelerate the ALS flow. Chapter 6 will demonstrate a low-power and high-speed approximate LUT architecture and its supporting synthesis algorithms. Finally, Chapter 7 will conclude this dissertation and discuss future works.

Chapter 2

ALSRAC: Area-Driven Approximate Logic Synthesis Flow by Resubstitution with Approximate Care Set

This chapter presents a novel area-driven ALS flow called ALSRAC that can generate high-quality approximate circuits.

2.1 Motivations and Overview

As mentioned in Section 1.4.1, many ALS methods apply LACs to approximately simplify circuits. Previous proposed LACs tend to have two drawbacks: they are either *too local* or *too coarse*. A LAC is too local if it approximates the node's function using only nearby nodes, such as direct fanins. More distant nodes are not used to express the approximate function, which may lose opportunities for better approximation. For instance, Wu and Qian (2016) introduced a LAC for Boolean networks, which removes literals from the Boolean expression of each node in the network. In this case, the approximate function is expressed with the node fanins. Yao et al. (2017) approximated a MFFC (defined in the reference (Cong et al., 1999)) in the circuit by a tree of gates obtained through approximate disjoint bi-decomposition. The new function is represented by the inputs of the MFFC. On the other hand, a LAC is too coarse if it fails to closely approximate the original function. Thus, large errors may occur. For instance, Shin and Gupta (2011) proposed to replace a gate by a constant zero or one. Chandrasekharan et al. (2016) rewrote the critical local cuts in an AIG with constant zero. Such

constant substitutions produce large errors in the circuit. Venkataramani et al. (2013) proposed SASIMI, where a LAC substitutes a node by another node with a similar function. Su et al. (2018) further improved SASIMI by an accurate and efficient batch error estimation approach. Yet, such a single-input substitution may also introduce large errors, since a function usually depends on multiple inputs, and the expressive power of a single input is limited.

To address the above issues of previous proposed LACs, this chapter attempts to find a LAC that replaces a node function with a multi-input function. It is neither too local if remote nodes are selected to express the new function, nor too coarse due to the rich expression power of multi-input functions. Notice that in traditional logic synthesis, such a replacement is called a *resubstitution*. A previous work generates resubstitutions using the care set of a node (Mishchenko et al., 2011). This chapter proposes to generate *approximate resubstitutions* using an approximate care set.

The main contributions of this chapter are as follows:

- This chapter proposes to approximate the care set using logic simulation. For scalability, the care patterns are expressed in terms of internal nodes instead of the PIs.
- This chapter proposes to generate approximate resubstitutions by computing *irredundant sums-of-products* (*ISOPs*) using approximate care patterns. This novel type of LAC exploits distant signals to form an approximation and has strong expressive power.
- This chapter proposes an ALS flow by resubstitution with approximate care set, *ALSRAC*. It is an efficient simulation-only logic synthesis flow that does not rely on complex Boolean manipulation engines, such as SAT and BDD.

ALSRAC is applicable to average error metrics such as error rate and mean error distance. The experimental results show that it improves the quality of approximate circuits significantly on various benchmarks and error metrics. It can efficiently generate approximate circuits and achieve 7%–18% more area savings than the state-of-the-art methods. As a byproduct of area reduction, ALSRAC also reduces the circuit delay.

The remainder of this chapter is organized as follows. Section 2.2 introduces the preliminaries. Section 2.3 elaborates the proposed LAC and the ALSRAC methodology. The experimental results are presented in Section 2.4. Finally, Section 2.5 summarizes the chapter.

2.2 Preliminaries

This section presents the preliminaries related to ALSRAC. Circuit terminologies and average error metrics will be introduced as follows.

2.2.1 Circuit Terminologies

This chapter focuses on multi-level combinational circuits, which can be modeled as a directed acyclic graph. In a circuit, a *primary input* (*PI*) is a node without any fanin. A *functional node* performs logic operations. A *primary output* (*PO*) is a dummy node driven by a functional node or a PI. It has a single fanin and no fanout. A *path* is a sequence of connected nodes in the circuit.

A circuit can be represented in various forms. In an *AND-inverter graph* (*AIG*), functional nodes are two-input AND gates, and edges can be either complemented or non-complemented, where a complemented edge indicates the negation of the signal. For example, Fig. 2.1 shows an AIG in which all the edges are non-complemented. It has 5 PIs $x_0, x_1, \ldots, x_4$, 6 functional nodes (two-input AND gates) $n_0, n_1, \ldots, n_5$, and 2 POs y_0 and y_1. In a *majority-inverter graph* (*MIG*), functional nodes are three-input majority nodes, and edges can also be either complemented or non-complemented. In a LUT network, which is the underlying representation for an FPGA design, functional nodes are LUTs. In a gate netlist, functional nodes are gates. For both LUT network and gate netlist, edges can only be non-complemented.

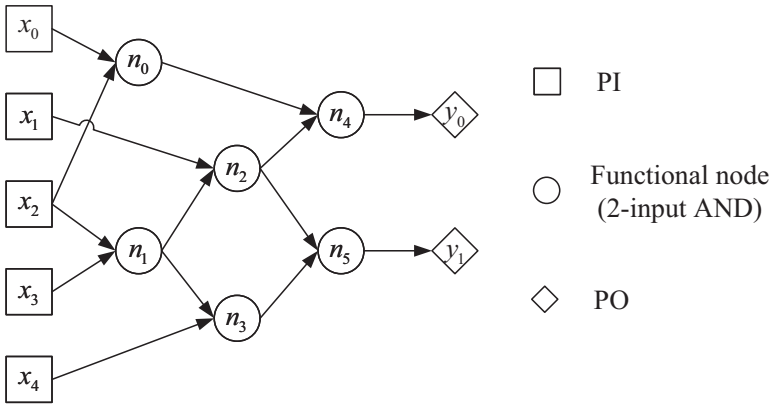


Figure 2.1: An example circuit in the AIG representation, where all the edges are non-complemented.

The inputs of a node are called its *fanins*. If there is a path from node A to B, A is a *transitive fanin* (*TFI*) of B. The *TFI cone* of a node includes the node itself and its TFIs (Mishchenko and Brayton, 2005). For example, in Fig. 2.1, the TFIs of n_2 are x_1, x_2, x_3, and n_1. The TFOs of n_1 are n_2, n_3, n_4, n_5, y_0, and y_1.

The *don't-cares* for a logic function are those input patterns that allow its output to be either 0 or 1. All

don't-care patterns constitute the *don't-care set* of the function, while the complement is the *care set*. Due to the complement relationship, the rest of the chapter only focuses on the care set.

An *irredundant sum-of-products* (*ISOP*) of a function is a sum-of-product expression, in which each product term is a prime implicant, and no product term can be deleted without changing the function (Sasao and Butler, 2001).

2.2.2 Average Error Metrics

Error metrics are essential for measuring the accuracy of approximate circuits. Among the various error metrics for evaluating the accuracy of approximate circuits, this chapter focuses on the commonly-used *average error metrics*. Let $\boldsymbol{y} : \mathbb{B}^I \to \mathbb{B}^O$ and $\hat{\boldsymbol{y}} : \mathbb{B}^I \to \mathbb{B}^O$ be the multiple-output Boolean functions of an exact circuit and an approximate circuit, respectively. Assume that the numbers of PIs and POs of the circuits are I and O, respectively. The *average error* is defined as the average deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ over all input patterns:

$$\text{average error} = \sum_{\boldsymbol{x} \in \mathbb{B}^I} \varepsilon(\boldsymbol{y}(\boldsymbol{x}), \hat{\boldsymbol{y}}(\boldsymbol{x})) \cdot p(\boldsymbol{x}),$$

where $\boldsymbol{y}(\boldsymbol{x})$ and $\hat{\boldsymbol{y}}(\boldsymbol{x})$ are binary vectors of length O, denoting the outputs of the exact and the approximate circuits under the input pattern $\boldsymbol{x}$, respectively, $p(\boldsymbol{x})$ is the *occurring probability* of the pattern $\boldsymbol{x}$, and ε is a *distance function* measuring the deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$.

Typical average errors include *error rate* (*ER*), *mean Hamming distance* (*MHD*), *mean error distance* (*MED*), and *mean relative error distance* (*MRED*). ER is the probability of an input pattern producing a wrong output for the approximate circuit. Its distance function is

$$\varepsilon_{\text{ER}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \begin{cases} 0, \text{if } \boldsymbol{y} = \hat{\boldsymbol{y}}, \\ 1, \text{if } \boldsymbol{y} \neq \hat{\boldsymbol{y}}. \end{cases}$$

ER is suitable for evaluating the accuracy of circuits such as classifiers, controllers, and error correctors. It is also widely used as an additional error metric for arithmetic circuits (Liu et al., 2014a).

MHD is the average number of bit-flips in $\hat{\boldsymbol{y}}$ *with respect to* (*w.r.t.*) the original $\boldsymbol{y}$. Its distance function is

$$\varepsilon_{\text{MHD}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{i=0}^{O-1} |y_i - \hat{y}_i| ,$$

where y_i and $\hat{y}_i \in \{0, 1\}$ denote the i-th PO of the exact and the approximate circuits, respectively. MHD is

often used as an additional error metric for arithmetic circuits.

MED is the mean absolute difference between the numerical values encoded by the outputs of exact and approximate circuits. Its distance function is

$$\varepsilon_{\mathrm{MED}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = |\mathit{int}(\boldsymbol{y}) - \mathit{int}(\hat{\boldsymbol{y}})|,$$

where the function $\mathit{int}(\boldsymbol{v})$ returns the integer encoded by the binary vector $\boldsymbol{v}$. MED can measure the accuracy of arithmetic circuits, such as adders and multipliers.

MRED is the mean of the relative error between the numerical values encoded by the outputs of exact and approximate circuits. Its distance function is

$$\varepsilon_{\mathrm{MRED}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{|\mathit{int}(\boldsymbol{y}) - \mathit{int}(\hat{\boldsymbol{y}})|}{\max\{\mathit{int}(\boldsymbol{y}), 1\}}.$$

To avoid division by zero, the denominator above is set as the maximum of the correct output value and 1. MRED can measure the accuracy of arithmetic circuits, such as adders and multipliers (Jiang et al., 2017).

Average errors are usually estimated with *Monte Carlo* simulation by sampling a set of M input patterns $\mathbb{X} = \{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^M\}$. That is,

$$\text{average error} \approx \frac{1}{M} \sum_{\boldsymbol{x} \in \mathbb{X}} \varepsilon(\boldsymbol{y}(\boldsymbol{x}), \hat{\boldsymbol{y}}(\boldsymbol{x})).$$

In addition, the *normalized* MHD (NMHD) and *normalized* MED (NMED) are defined as follows:

$$\mathrm{NMHD} = \frac{\mathrm{MHD}}{O}, \mathrm{NMED} = \frac{\mathrm{MED}}{2^O - 1}.$$

2.3 Methodology

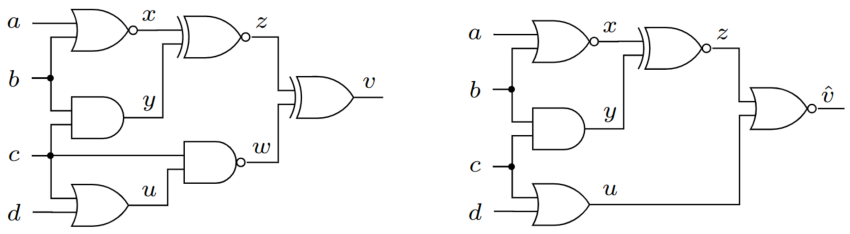
This section presents the technique to generate the approximate care set. Then, the proposed LAC is introduced, which is a resubstitution using this care set. Finally, an ALS flow based on the LAC is presented.

2.3.1 Approximation of Care Set

The set of approximate care patterns for a node function is generated using logic simulation with random input patterns following a user-specified distribution.

Assume that a node function is to be expressed using a set of nodes belonging to the same circuit, called *divisors*. The patterns appearing at the divisors after logic simulation are called the *approximate cares of the node at the divisors*. All PI patterns producing the approximate cares at the divisors are called the *approximate cares of the node at the PIs*. In what follows, unless otherwise specified, we use *approximate cares* to refer to approximate cares at the divisors. Since only some patterns are selected as the approximate cares, such an approximation shrinks the care set. The following example shows how to generate an approximate care set.

Example 2.1. The circuit shown in Fig. 2.2(a) has 4 inputs a, b, c, and d, with 16 PI combinations in total. The node values under all PI patterns are listed in Table 2.1. To derive an approximate function of node v with divisors $\{u, z\}$, 5 PI patterns are randomly selected, *i.e.*, $abcd = \{0000, 0010, 0011, 0100, 1000\}$ (see the shaded rows in Table 2.1), and the circuit is simulated. The patterns $\{00, 01, 10\}$ appearing at divisors $g = \{u, z\}$ are the approximate cares of node v at g. Among all the 16 PI combinations, 11 patterns produce the approximate cares at g. They are the approximate cares of node v at the PIs. Based on this, v's function can be simplified from the original expression $v = z \oplus w$ to $\hat{v} = \neg(u \vee z)$ (see Example 2.4 for details). The resulting circuit is shown in Fig. 2.2(b), which is simplified compared to the original one. If the inputs are uniformly distributed, such a simplification introduces 18.75% error rate at node v (*i.e.*, 3 in 16 PI patterns generate incorrect outputs).



(a) Original circuit. **(b)** Approximate circuit.

Figure 2.2: Example circuits.

Table 2.1: Node values under all PI patterns for the circuit in Fig. 2.2(a).

abcd	x	y	**u**	**z**	w	v	*abcd*	x	y	**u**	**z**	w	v
0000	1	0	**0**	**0**	1	1	1000	0	0	**0**	**1**	1	0
0001	1	0	**1**	**0**	1	1	1001	0	0	**1**	**1**	1	0
0010	1	0	**1**	**0**	0	0	1010	0	0	**1**	**1**	0	1
0011	1	0	**1**	**0**	0	0	1011	0	0	**1**	**1**	0	1
0100	0	0	**0**	**1**	1	0	1100	0	0	**0**	**1**	1	0
0101	0	0	**1**	**1**	1	0	1101	0	0	**1**	**1**	1	0
0110	0	1	**1**	**0**	0	0	1110	0	1	**1**	**0**	0	0
0111	0	1	**1**	**0**	0	0	1111	0	1	**1**	**0**	0	0

In this chapter, approximation of the care set is expressed by representing care patterns at the set of divisors. Compared to the previous approaches based on approximate *external don't-cares* (*EXDCs*) (Venkataramani et al., 2012; Miao et al., 2014), the proposed representation scales better for large circuits since EXDCs are expressed at the PIs. Indeed, in Example 2.1, the 3 approximate cares at divisors $\{u, z\}$ correspond to 11 approximate cares at the PIs. It implies that a few approximate cares expressed with divisors are equivalent to many approximate cares expressed with PIs. Therefore, even in large circuits with many PIs, the approximate care patterns can be stored without a large memory overhead, while they have the same expressive power as the approximate cares at the PIs.

2.3.2 Proposed Local Approximate Change

This section presents a novel LAC by resubstituting a node function with another function derived using the approximate care set. The original node function is represented with its fanins, while the *approximate resubstitution function* is expressed with some divisors that are not necessarily its fanins. In Example 2.1, the original node function for v is $z \oplus w$. It is replaced by divisors $\{u, z\}$ with a resubstitution function $v = \neg(u \vee z)$, using the approximate care set $uz = \{00, 01, 10\}$.

To generate approximate resubstitutions for a node, three basic questions need to be answered:

- How to select good divisors among internal nodes in the circuit?
- Is a given set of divisors feasible to perform approximate resubstitution of the function?
- Given a feasible divisor set and an approximate care set, how to derive an approximate resubstitution function of the node, which can replace the original function?

These three questions are answered one by one in the following paragraphs. After that, the method to generate LAC candidates is presented.

Selection of Divisors

For each node, there are numerous divisor sets producing different resubstitutions. Only part of them are selected for the sake of efficiency. The proposed strategy is illustrated in Algorithm 1. Specifically, the divisor sets for a node V are generated by the following operations.

- Remove a fanin n from the fanin set of node V (Lines 5–5).
- Replace a fanin n in the fanin set of node V with another node u in V's TFI cone (Lines 6–8).

Only the divisors in V's TFI cone are considered since the function of V more likely depends on them, instead of the divisors outside the cone.

Algorithm 1: *Select_Divisor_Sets*(V, G)

Input: node V, circuit G.
Output: divisor sets S.

```
1 Find the TFI cone T of node V in circuit G;
2 Sort nodes in T in ascending order of logic levels;
3 Divisor sets S ← ∅, node set FI ← V's fanins;
4 for each node n in FI do
      // remove a fanin n
5     Divisor set A ← FI\{n}; Add A into S;
6     for each node u in TFI cone T do
          // replace a fanin n by u
7         Divisor set B ← A ∪ {u};
8         Add B into S;
9 return S
```

Feasibility of Divisors

Note that given a set of divisors, it may be impossible to derive a function with them to substitute the original node function. For instance, in Fig. 2.2(a), a function in terms of a and b to resubstitute v cannot be found. The reason is that v not only depends on a and b, but also on c and d. Thus, for a divisor set generated from Algorithm 1, it is necessary to check the feasibility of the divisors to form a valid resubstitution function. The proposed check method is based on the following theorem for checking the feasibility of an *accurate* resubstitution (Mishchenko et al., 2006).

Theorem 2.1. *Assume that there are m divisors with functions $g_1(\mathbf{x})$, $g_2(\mathbf{x})$, ..., $g_m(\mathbf{x})$ and a node V with the function $F(\mathbf{x})$, where $\mathbf{x}$ is the set of PI variables. The divisors can form an accurate resubstitution function of node V, if and only if there are no PI patterns $\mathbf{x}_1$ and $\mathbf{x}_2$, such that $F(\mathbf{x}_1) \neq F(\mathbf{x}_2)$, but $g_j(\mathbf{x}_1) = g_j(\mathbf{x}_2)$ for all $1 \leq j \leq m$.*

The theorem can be explained intuitively as follows. If there is a function $H(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$ that can accurately resubstitute $F(\mathbf{x})$, then for any PI patterns $\mathbf{x}_1$ and $\mathbf{x}_2$ satisfying $g_j(\mathbf{x}_1) = g_j(\mathbf{x}_2)$ for all j's, we must have $F(\mathbf{x}_1) = H(g_1(\mathbf{x}_1), \ldots, g_m(\mathbf{x}_1)) = H(g_1(\mathbf{x}_2), \ldots, g_m(\mathbf{x}_2)) = F(\mathbf{x}_2)$. A simple application of Theorem 2.1 is as follows.

Example 2.2. For the circuit in Fig. 2.2(a), to check whether divisors $\{u, z\}$ can accurately resubstitute v with a function, all 16 PI patterns are enumerated. Under PI patterns $abcd = 0001$ and 0010, node v takes different

values. Yet the patterns on $\{u, z\}$ are the same (*i.e.*, $uz = 10$). By Theorem 2.1, it is impossible to derive a function in terms of $\{u, z\}$ to accurately resubstitute v.

In the reference (Mishchenko et al., 2006), Theorem 2.1 is converted into a SAT instance to check the feasibility of divisors. It is time-consuming for large circuits. For approximate computing, it is not necessary to check all the PI combinations with a SAT solver. Instead, only PI patterns appearing in the logic simulation are checked. If Theorem 2.1 is satisfied under PI patterns appearing in limited simulation rounds, then the given divisors can form an approximate resubstitution function.

Example 2.3. Consider the circuit in Fig. 2.2(a). Assume that the same PI patterns of Example 2.1, *i.e.*, $abcd = \{0000, 0010, 0011, 0100, 1000\}$ (see the shaded rows in Table 2.1), are selected to perform logic simulation. It needs to be checked whether divisors $\{u, z\}$ can resubstitute v with an approximate function under these simulation patterns. In this case, the corresponding patterns on the divisor set are $uz = \{00, 10, 10, 01, 01\}$, while v takes the value $\{1, 0, 0, 0, 0\}$. Note that each pattern on $\{u, z\}$ corresponds to only one value of v (*i.e.*, 00, 01, and 10 correspond to 1, 0, 0, respectively). Thus, Theorem 2.1 holds for the set of random simulation patterns. Therefore, it is possible to derive an approximate function in terms of $\{u, z\}$ to resubstitute v. In other words, $\{u, z\}$ is a feasible divisor set.

Derivation of Approximate Resubstitutions

At this point, feasible divisor sets have been selected. The approximate cares on each divisor set are also obtained. Approximate resubstitution functions will be derived from them. Instead of utilizing a SAT-based method in the reference (Mishchenko et al., 2011) or a BDD-based method in the reference (Mishchenko et al., 2006), approximate resubstitution functions can be generated efficiently by computing ISOP expressions.

For node V and a feasible divisor set g of V, the truth table of an approximate resubstitution function for V on the approximate cares at g is built. Its input variables are the nodes in g and the output variable is node V. In the truth table, if an input combination is not in the approximate care set at nodes in g, its output value is a don't-care ("$-$"). Otherwise, the output value is the value of V for the corresponding approximate care pattern. Note that since the divisor set is feasible, although there may be many PI patterns producing the same approximate care pattern, the value of node V for all of them is the same.

Next, an ISOP expression is computed from the truth table as the approximate resubstitution function using Espresso (Mcgeer et al., 1993). After that, the ISOP expression will be converted to some nodes in the circuit. Then, the new nodes are used to replace the original node V and change the local structure of the circuit.

Example 2.4. From Example 2.3, for the circuit in Fig. 2.2(a), there exist approximate functions in terms of $\{u, z\}$ to resubstitute v if logic simulation is performed with 5 PI patterns $abcd = \{0000, 0010, 0011, 0100, 1000\}$. To derive an approximate function, a truth table shown in Table 2.2 is built with inputs u and z and output $\hat{v}$. The pattern $uz = 11$ is not in the approximate care set, so it is a don't-care pattern. For the approximate care patterns $uz = \{00, 01, 10\}$, the corresponding output values are set as $\hat{v} = \{1, 0, 0\}$, which can be directly obtained from Table 2.1. If $\hat{v}$ takes 0 when $uz = 11$, a possible ISOP expression is $\hat{v} = \neg u \wedge \neg z$, and it is converted into a NOR gate. Then, gates w and v in Fig. 2.2(a) are removed, and NOR gate $\hat{v}$ is added into the circuit, producing the approximate circuit shown in Fig. 2.2(b).

Table 2.2: A truth table of an approximate function with inputs u and z and output $\hat{v}$ in Example 2.4.

uz	00	01	10	11
$\hat{v}$	1	0	0	–

Generation of LAC Candidates

After showing the answers to the three basic questions, the method to generate LAC candidates is presented. It is shown in Algorithm 2.

Algorithm 2: *Generate_LACs*(G, N, L)

Input: circuit G, simulation round N, limit of LAC count L.
Output: LAC candidate set Π.

```
1  Perform logic simulation for N rounds;
2  LAC candidate set Π ← ∅;
3  for each node V in G do
4  |  Divisor sets S ← Select_Divisor_Sets(V, G);
5  |  LAC count l ← 0;
6  |  for each divisor set g in S do
7  |  |  if l ≥ L then break;
8  |  |  if g is feasible to resubstitute V then
9  |  |  |  l ← l + 1;
10 |  |  |  Build truth table B with input set g and output V;
11 |  |  |  New function f ← Find_ISOP(B);
12 |  |  |  Add function f into candidate LAC set Π;
13 return Π
```

The inputs of Algorithm 2 are a circuit G, a simulation round N, and the maximum number of LAC candidates L, while its output is a LAC candidate set Π. First, N rounds of logic simulation are performed in circuit G (Line 1). Then, for each node, some divisor sets S are selected by Algorithm 1 (Line 4). After that, the feasibility of each divisor set (Line 8) is checked. If it is feasible, the truth table B with input set

g and output V are generated (Line 10). Based on this, an approximate resubstitution function f is derived by computing an ISOP expression and added into the LAC candidate set Π (Lines 11–12). To limit the total number of LACs, at most L LACs are generated for each node V (Line 7).

2.3.3 Proposed ALS Flow

This section presents ALSRAC, an ALS flow based on the proposed LACs, which is shown in Algorithm 3.

Algorithm 3: ALSRAC flow.

Input: original circuit G_{in}, error threshold E_t, initial simulation round N, limit of LAC count L, controlling parameter t, scaling factor r $(0 < r < 1)$.
Output: approximate circuit G_{out}.

```
1  Circuit G ← Convert_To_AIG(G_in), E ← 0;
2  while E ≤ E_t do
3      Generate N PI patterns randomly;
4      Candidate LAC set Π ← Generate_LACs(G, N, L);
5      if Π ≠ ∅ then
6          Find the best LAC in Π with the smallest error E;
7          if E > E_t then break;
8          Apply the best LAC with the smallest error to G;
9          Optimize G with traditional logic synthesis tool;
10     if Π = ∅ for continuous t iterations then N ← N × r;
11 G_out ← Technology_Mapping(G);
12 return G_out
```

The inputs of the algorithm are an accurate circuit G_{in}, an error threshold E_t, an initial simulation round N, a limit of LAC count L, a controlling parameter t, and a scaling factor r. The controlling parameter and the scaling factor are related to the adjustment of N, which will be described later. The flow works on an AIG iteratively and returns an approximate design G_{out} meeting the error threshold. For each iteration, if the error is no more than the threshold, the LAC candidates are generated by Algorithm 2 (Line 4). Then, the induced errors of LACs are evaluated and the best LAC with the smallest error E is found (Line 6). In order to evaluate the errors of all LACs efficiently, the batch error estimation method is applied (Su et al., 2018), which has the same error estimation accuracy as applying traditional simulation to each LAC individually, but is much faster. If the error E does not exceed the error threshold E_t (Line 7), the best LAC is applied to simplify the circuit (Line 8). Due to the redundancies in the circuit after applying the LAC, traditional logic optimization is performed (Line 9). Finally, after the iteration loop terminates, technology mapping is performed and the resulting approximate circuit satisfying the error constraint is returned (Line 11).

One important feature of ALSRAC is that the simulation round N is adjusted dynamically to control the approximation degree. Theoretically, as N increases, the approximate care set approaches the accurate

one. In this case, the LAC will be closer to the accurate function, and hence, the induced error of the LAC decreases. However, since a large N increases the size of the approximate care set, it limits the approximation space. Therefore, sometimes finding a LAC from approximate care patterns with a large N is difficult. On the contrary, as N decreases, it is easier to find a LAC.

In real implementation, N is set to a suitable value initially. If the circuit to be approximated does not have any LAC candidates (*i.e.*, $\Pi = \emptyset$), it means that the size of the care set is too large. In this case, the approximate care set by reducing the simulation round N should be shrunk. Thus, more LAC candidates with larger induced errors can be generated. However, different PI patterns generate distinct approximate care patterns. Therefore, when $\Pi = \emptyset$ for a set of PI patterns, another set of PI patterns to get a different care set can be tried (Line 3), which may generate some LACs. Hence, N will not be reduced as soon as $\Pi = \emptyset$. Instead, a controlling parameter t is introduced to control the decreasing of simulation round N. N will be scaled by r ($0 < r < 1$) only when $\Pi = \emptyset$ for t consecutive iterations (Line 10).

2.4 Experimental Results

This section presents the experimental results of ALSRAC.

2.4.1 Experiment Setup

ALSRAC is implemented in C++ and tested on a single-core AMD 3700X processor running at 3.8GHz with 16GB RAM. ER and ED are selected as the error metrics, which are measured by performing 10,000,000 rounds of logic simulation to guarantee good accuracy. All PI vectors are assumed to be uniformly distributed, although the proposed method is applicable to any PI distribution. The *area ratio* (the area of the approximate circuit over the original one) and the *delay ratio* (the delay of the approximate circuit over the original one) are used to evaluate the approximated designs. Both ASIC and FPGA designs are considered. Particularly, the area and delay of an FPGA design are measured using its LUT count and the depth of its LUT network, respectively. It is obvious that smaller ratios are preferred due to more reduction in area and delay. The traditional optimization (Line 9 in Algorithm 3) is performed by ABC (Mishchenko et al., 2022) using commands "*sweep; resyn2*".

Important parameters of the proposed method are listed below. The number of simulation rounds N is initially set to 32. The limit on LAC counts at each node is $L = 1$. The simulation round N is scaled by $r = 0.9$ only if there are no LACs for successive $t = 5$ iterations.

It seems that the initial simulation round $N = 32$ is negligible compared to the number of all PI patterns,

which possibly introduces large errors in the circuit. However, it does generate good approximate circuits in the experiments even for a small error threshold. The reason is that ALSRAC works on an AIG (Line 1 in Algorithm 3). The care sets and the resubstitution functions are expressed with at most 2 divisors. Thus, even using just a few PI patterns, the approximate care set is still close to the accurate one.

In addition, due to the randomness of logic simulation, ALSRAC may produce different approximate circuits in different runs. The experiments have been performed three times and the average circuit quality and runtime are reported.

Table 2.3: Benchmarks used in experiments.

ISCAS & arithmetic			EPFL random/control			EPFL arithmetic		
Circuit	Area	Delay	Circuit	#LUT	Depth	Circuit	#LUT	Depth
alu4	2798	12.7	arbiter	409	23	adder	192	64
c1908	758	37.3	cavlc	101	6	shifter	512	4
c2670	1262	21.9	alu ctrl	27	2	divisor	3268	1208
c3540	1604	55.0	decoder	270	2	hyp	40406	4532
c5315	2451	47.5	i2c ctrl	227	7	log2	6574	119
c7552	2756	69.4	Int2float	28	6	max	523	189
c880	585	24.9	mem ctrl	2354	22	mult	4923	90
cla32	958	38.5	priority	110	26	sine	1229	55
ksa32	1128	17.8	router	52	6	sqrt	3077	1106
mtp8	1069	37.8	voter	1301	17	square	3246	74
rca32	666	16.1						
wal8	1081	45.3						

2.4.2 Experiments on ASIC Designs

This experiment compares ALSRAC with a state-of-the-art ALS approach, Su's method (Su et al., 2018), on ASICs under ER and NMED constraints. Both methods aim at the best approximate circuits with the smallest area ratio satisfying the error constraint. The tested benchmarks are some ISCAS and arithmetic benchmarks. They are listed in the first column of Table 2.3 and are optimized with the logic synthesis tool SIS (Sentovich et al., 1992) before conducting the experiments. They are the same ones as those used in the reference (Su et al., 2018). Su's method is reimplemented with C++. The final approximate circuits in Su's method and ALSRAC are mapped with MCNC standard cell library (Yang, 1991) using the ABC command "*map -D <original delay>*".

Comparison under ER constraint

This experiment compares ALSRAC with Su's method under ER constraint using the ISCAS and the arithmetic circuits. The area ratio, delay ratio, and runtime are listed in Table 2.4. The values for each benchmark are the

Table 2.4: Comparison of ALSRAC and Su's method under ER constraint.

Circuit	Average area ratio		Average delay ratio		Average time (s)	
	ALSRAC	Su's	ALSRAC	Su's	ALSRAC	Su's
alu4	**70.46%**	73.72%	101.35%	100.00%	**547**	10639
c1908	**75.24%**	77.33%	79.70%	76.98%	**19**	398
c2670	**66.75%**	74.60%	**89.89%**	97.46%	**6**	491
c3540	**92.89%**	94.66%	**86.47%**	100.00%	**199**	2063
c5315	**87.91%**	95.50%	**73.53%**	99.07%	**66**	3192
c7552	**80.30%**	91.13%	**77.97%**	94.96%	**153**	10275
c880	**90.48%**	93.58%	89.50%	87.49%	**20**	59
cla32	**59.72%**	79.69%	84.34%	58.66%	**6**	625
ksa32	**70.17%**	84.14%	91.17%	72.79%	**91**	1148
mtp8	**95.31%**	96.73%	**91.61%**	98.94%	**30**	548
rca32	**91.31%**	94.79%	99.56%	99.47%	**5**	28
wal8	**80.80%**	93.56%	95.90%	82.09%	**9**	1092
Arithmean	**80.11%**	87.45%	**88.42%**	88.99%	**32**	2546

average results under 7 ER thresholds (0.1%, 0.3%, 0.5%, 0.8%, 1%, 3%, 5%). The entries in **bold** highlight the cases where ALSRAC is better than Su's method. Similar notations are applied in the following tables. For all cases, ALSRAC reduces more area than Su's method. On average, ALSRAC generates approximate circuits with an area ratio of **80.11**%, improving over Su's method by **8.39**% relatively. For *c7552*, *cla32*, *ksa32*, and *wal8*, ALSRAC reduces more than **10**% area over Su's method. Furthermore, all approximate designs other than *alu4* generated by ALSRAC have smaller delays compared to the accurate circuits. Nearly half of the approximate designs have better area and delay simultaneously than the circuits produced by Su's method. With regard to the runtime, ALSRAC is **80**× faster than Su's method on average.

Comparison under NMED constraint

This experiment compares ALSRAC and Su's methods under the NMED constraint. Only the arithmetic circuits are selected since NMED is an error metric for arithmetic circuits. Table 2.5 lists the area ratio, delay ratio, and runtime under the NMED constraint. The results for each benchmark are the averages under 8 NMED thresholds (0.00153%, 0.00305%, 0.00610%, 0.01221%, 0.02441%, 0.04883%, 0.09766%, 0.19531%). ALSRAC always reduces more area than Su's method. On average, ALSRAC produces approximate circuits with an area ratio of **39.64**%, improving over Su's method by **18.15**% relatively. All approximate circuits generated by ALSRAC have smaller delays than the accurate ones. Two of them have better area and delay at the same time than the circuits generated by Su's method. Additionally, ALSRAC has a speed-up of **3**× over Su's method on average.

Table 2.5: Comparison of ALSRAC and Su's method under NMED constraint.

Circuit	Average area ratio		Average delay ratio		Average time (s)	
	ALSRAC	Su's	ALSRAC	Su's	ALSRAC	Su's
cla32	**15.85%**	26.03%	58.34%	52.73%	**404**	2723
ksa32	**16.11%**	26.16%	87.78%	78.86%	**714**	4498
mtp8	**78.60%**	82.65%	**97.52%**	97.62%	**2044**	2254
rca32	**23.48%**	28.68%	**86.57%**	95.81%	**514**	706
wal8	**64.14%**	78.63%	90.70%	89.74%	**758**	3592
Arithmean	**39.64%**	48.43%	84.18%	82.95%	**887**	2754

2.4.3 Experiments on FPGA Designs

This experiment compares ALSRAC with a state-of-the-art ALS approach, Liu's method (Liu and Zhang, 2017), on FPGAs under ER and MRED constraints. Both methods aim at finding the best approximate circuit with the smallest area ratio satisfying the error constraint. The EPFL benchmarks are used to test the performance on FPGA designs (EPFL LSI Lab, 2021). They are listed in the last two columns of Table 2.3, which show the best synthesis results obtained in recent years. The final approximate circuits are mapped into 6-LUTs using the ABC command "*if -K 6*".

Comparison under ER constraint

This experiment compares ALSRAC with Liu's method using EPFL random/control benchmarks using the ER threshold of 1%. The area ratio, delay ratio, and runtime are listed in Table 2.6. In all cases, ALSRAC reduces more area than Liu's method. On average, ALSRAC generates approximate circuits with an area ratio of **74.30**%, improving over Liu's method by **7.41**%. Meanwhile, the average delay ratio is improved by **10.60**%. In particular, only **9.09**% of LUTs are needed to approximate the *priority* circuit. All approximate circuits generated by ALSRAC have delays no larger than the accurate ones. Half of them have both better area and delay compared to Liu's designs. Only the runtime of ALSRAC is provided since that of Liu's method is not mentioned in the reference (Liu and Zhang, 2017). ALSRAC is efficient for all cases, even for the largest circuit *mem ctrl* with 2354 6-LUTs, ALSRAC can produce an approximate design in an acceptable time.

Comparison under MRED constraint

This experiment compares ALSRAC with Liu's method using EPFL arithmetic benchmarks under the MRED threshold of 0.19531%. Notably, Liu's work uses MRED instead of NMED as the measure of ED. Thus, this experiment also uses MRED as the error metric. The area ratio, delay ratio, and runtime are listed in Table 2.7. All the EPFL arithmetic circuits are listed in the table except *hyp* since it has a massive number (40406) of

Table 2.6: Comparison of ALSRAC and Liu's method under ER constraint.

Circuit	Area ratio		Delay ratio		ALSRAC
	ALSRAC	Liu's	ALSRAC	Liu's	time (s)
arbiter	**53.06%**	61.37%	**43.48%**	56.52%	39
cavlc	**93.07%**	99.01%	83.33%	83.33%	69
alu ctrl	**96.30%**	100.00%	100.00%	100.00%	0.2
decoder	**97.78%**	100.00%	100.00%	100.00%	5
i2c ctrl	**78.41%**	90.31%	85.71%	85.71%	24
Int2float	**89.29%**	92.86%	**83.33%**	100.00%	4
mem ctrl	**70.56%**	88.62%	**36.36%**	68.18%	2059
priority	**9.09%**	10.91%	11.54%	11.54%	2
router	**57.69%**	59.62%	**16.67%**	33.33%	2
voter	**97.77%**	99.85%	**99.85%**	100.00%	967
Arithmean	**74.30%**	80.25%	**66.03%**	73.86%	317

6-LUTs and ALSRAC cannot synthesize it within 24 hours. In terms of performance, ALSRAC reduces more area than Liu's method for all the benchmarks except *divisor* and *max*. Especially for the circuit *max*, ALSRAC is not competitive compared to Liu's method. With/without the *max* circuit, ALSRAC generates approximate designs with an area ratio of **59.69**%/**56.20**%, improving by **1.48**%/**11.86**% compared to Liu's method. In particular, for *shifter* and *mult*, ALSRAC reduces nearly **20**% more area. Besides, only **3.09**% of 6-LUTs are required to approximate the *sqrt* circuit. All approximate circuits generated by ALSRAC have delays no larger than the accurate ones. More than half of them have both areas and delays no worse than Liu's method. The runtime is provided only for ALSRAC because the runtime is not reported in the reference (Liu and Zhang, 2017). Overall, the runtime of ALSRAC is acceptable.

Table 2.7: Comparison of ALSRAC and Liu's method under MRED constraint.

Circuit	Area ratio		Delay ratio		ALSRAC
	ALSRAC	Liu's	ALSRAC	Liu's	time (s)
adder	**68.23%**	71.35%	**4.69%**	15.63%	891
shifter	**71.48%**	90.04%	100.00%	100.00%	435
divisor	99.72%	98.90%	**70.94%**	88.41%	1982
log2	**90.07%**	97.37%	90.76%	90.76%	614
max	87.57%	35.18%	100.00%	10.05%	134
mult	**8.00%**	27.16%	**21.11%**	40.00%	25086
sine	**96.75%**	99.19%	**83.64%**	98.18%	52
sqrt	**3.09%**	10.98%	**1.81%**	10.13%	4238
square	**12.26%**	15.10%	27.03%	25.68%	9998
Arithmean	**59.69%**	60.59%	55.55%	53.20%	4826
Arithmean w/o max	**56.20%**	63.76%	50.00%	58.60%	5412

2.5 Summary

This chapter proposes ALSRAC, an area-driven ALS flow that relies on resubstitution with an approximate care set to produce high-quality approximate circuits. The main idea is to control the size of the approximate care set by logic simulation and find approximate changes that can be applied to the design. The experimental results show that ALSRAC can generate high-quality approximate circuits with significantly reduced area at the cost of small errors.


20006301

Chapter 3

Advanced Ordering Search Techniques for Approximate Logic Synthesis

This chapter presents advanced ordering search techniques for ALS, which can enhance the ability of ALS tools in area reduction and improve the quality of synthesized approximate circuits.

3.1 Motivations and Overview

As discussed in Section 1.4.1, most existing ALS methods are iterative. In each iteration, they determine which LAC should be applied greedily (Shin and Gupta, 2011; Venkataramani et al., 2013; Wu and Qian, 2016; Chandrasekharan et al., 2016; Yao et al., 2017). Specifically, a score is calculated for each valid LAC based on its local quality improvement (such as area improvement) and induced error. Then, the one with the highest score is selected. The iteration terminates when a given error limit is reached.

However, a greedy ALS method has its natural drawback of getting stuck into a local optimum. Fig. 3.1 shows an example for this. Assume that the error rate threshold is 1%. The target is to synthesize a circuit with the minimum area and error rate no more than 1%. Node ① is the input accurate circuit, which has an area of 305 and no error (*i.e.*, $A = 305, E = 0$). Each edge corresponds to a LAC and its score S is put near the edge. If the method always chooses a LAC with the highest score and modifies the circuit with the choice each time, it will move along the red path and generate an inexact design with area of 282 and error rate of 0.98%. Unfortunately, the best order leading to the best solution should be ①→②→④→⑦. Thus, a greedy ALS method does not guarantee to produce an approximate circuit with the best quality.

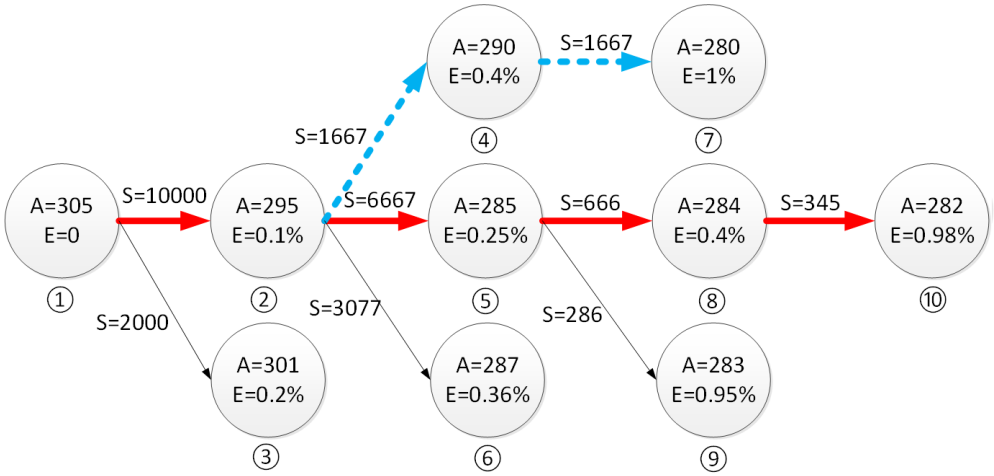


Figure 3.1: A greedy ALS method is stuck into a local optimum. A node represents a circuit and an edge represents a LAC. A and E denote the area and error rate of the corresponding circuit, respectively. S is the score of a LAC.

To overcome this issue, it is proposed to utilize two advanced search algorithms to find a better ordering of applying LACs.

The main contributions of this chapter are as follows:

- This chapter proposes an ordering search algorithm based on *beam search*. As an extension of the basic greedy method, beam search keeps track of multiple top-scored LACs and hence, explores a larger space for circuit simplification.

- This chapter further proposes an ordering search algorithm based on *Monte Carlo tree search* (MCTS). As a method for finding optimal decisions by taking random samples in the search space (Balla and Fern, 2009; Bjarnason et al., 2009; Mansley et al., 2011), MCTS is applied to the ALS problem to find a better ordering to apply LACs.

The experimental results show that these algorithms are effective in finding a better ordering than the basic greedy search and hence, improve the quality of the final approximate circuits.

The rest of this chapter is organized as follows. Section 3.2 presents the preliminaries. Section 3.3 elaborates the methodology of the proposed advanced ordering search algorithms. Section 3.6 shows the experimental results. Section 3.7 summarizes the work.

3.2 Preliminaries

This section presents the preliminaries related to this chapter. For the circuit terminologies and average error metrics, please refer to Section 2.2. Next, an important LAC used in this chapter will be introduced as follows.

The SASIMI LAC (Venkataramani et al., 2013) is used to illustrate the proposed method, although other types of LACs can also be applied. The SASIMI LAC replaces one signal in the circuit, called *target signal* (*TS*), by another, called *substitution signal* (*SS*). By doing so, the gates in the netlist only contributing to generating the TS can be removed, and the circuit area is reduced. In the approximate computing context, we do not require TS and SS to be exactly identical in their functions. A *valid* LAC in a gate netlist must satisfy:

- TS is a gate.
- SS can be a gate or its negation, a PI signal or its negation, or a constant 0/1.
- SS has no larger arrival time than TS. This guarantees that the final approximate circuit has no larger delay than the original accurate one.
- After substituting TS with SS, the ER conforms to the constraint.

3.3 Methodology

This section introduces the methodology of the proposed advanced ordering search methods.

Given an accurate gate netlist and an ER constraint, it is expected to find a good ordering to apply LACs, with which the ALS method could achieve an approximate circuit with smaller area. This problem can be formulated as a state-space search problem where states represent the original circuit and all approximate circuits obtained from it by successive applications of LACs. A good solution satisfying the ER constraint can be found by searching a corresponding state-space search tree. Fig. 3.1 shows an example of the search tree for the ALS problem. In such a tree, each node represents a gate netlist, and the actions of each node are valid LACs. The successors of a node represent all the circuits obtained by applying valid LACs on it. The root node is the original netlist and a leaf node of the tree, called *goal state*, is a circuit that cannot be further approximated without increasing its ER above the limit. The target is to achieve a goal state with the minimum area while satisfying the ER constraint.

The main challenge of the studied problem is the huge search space. On the one hand, by the type of LACs used here, the number of valid LACs at each node is quadratic to the number of signals in the circuit, since a TS can be any gate and an SS can be any signal with no larger arrival time. Consequently, the number of branches

from each node is extremely large. On the other hand, even if the branching factor is a small constant, the size of the search tree would grow exponentially with its depth. As many LACs may be applied before reaching the final ER limit, the depth of the search tree may be large. Therefore, it is infeasible to perform basic search algorithms such as breadth-first search and depth-first search.

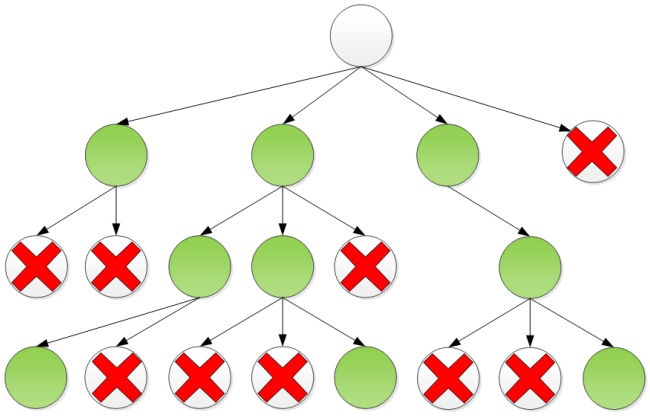


Figure 3.2: An illustration of beam search with $K = 3$.

To solve this challenge, a feasible way is to utilize advanced state-graph search algorithms. By focusing on promising candidates, these search algorithms tend to explore earlier the branches possibly containing the best solution in the search tree. Two advanced search algorithms are applied in this chapter. The first one is beam search, and the second is MCTS.

3.4 ALS by Beam Search

Beam search is a straightforward way to extend the basic greedy search. The basic greedy search only keeps one most promising state at each level of the search tree and expands it to reach to the states at the next level. Beam search extends this by always keeping K rather than just one most promising states at each level and expanding them to reach all new states (Russell, 2010). Fig. 3.2 illustrates a beam search with $K = 3$. At each level, all states expanded from the last level are checked and only 3 states (*i.e.*, the green nodes) with the highest scores are kept. Then, only these 3 states are expanded by their actions to reach the states at the next level.

The basic idea behind beam search is that it maintains K parallel "search threads", among which useful information is passed. In essence, the states that generate the best successors will inform the others that the optimal solution is more likely located in the corresponding branch. Thus, it quickly abandons unpromising searches and focuses on promising choices.

Algorithm 4: The BS-ALS algorithm.

Input : the original netlist C_{ori}, the error rate threshold T_{er}, and the branching factor K.
Output: an approximate netlist C_{ax}.

```
1  List L ← {C_ori}, set of candidate approximate circuits S ← ∅;
2  while L ≠ ∅ do
3      if |L| > K then
4          Keep K best circuits with the smallest error rates and remove others from L;
5      New list L_new ← ∅;
6      for each circuit C in L do
7          if C has no valid LACs then
8              S ← S ∪ C;
9          else
10             for each valid LAC l_v of C do
11                 apply l_v on C and get new circuit C_new;
12                 L_new ← L_new ∪ C_new;
13     L ← L_new;
14 Find the circuit C_ax with minimum area from S;
15 return C_ax
```

By combing Beam Search with the ALS method, the *BS-ALS* algorithm is designed. The detailed algorithm is shown in Algorithm 4. It takes the original circuit C_{ori} as input. A list L is maintained to store the best K states. Initially, L only contains the original circuit C_{ori} (Line 1). Meanwhile, a set S of candidate approximate circuits is used to record the possible solutions. In each iteration, the algorithm first resizes the list L and keeps at most K best circuits in L (Lines 3–4). In real implementation, ER is used as the criterion. Then, it creates an empty list L_{new} and iterates over all circuits in L. For a circuit C in L, if it has no valid LACs, it is a goal state and it will be added into the candidate set S (Lines 7–8). Note that by definition, a valid LAC should ensure that the ER of the updated circuit does not exceed the ER threshold. Thus, it is possible for a circuit to have no valid LACs. Otherwise, all circuits generated by applying the valid LACs to circuit C will be obtained and added to the list L_{new} (Lines 10–12). After all circuits in L are checked, L is updated by the list L_{new} (Line 13). The entire loop terminates until L is empty. This happens when all circuits in the last iteration have no valid LACs. Then, the circuit with the minimum area in set S is returned.

3.5 ALS by Monte Carlo Tree Search

MCTS is an emerging search method that exploits randomness to efficiently explore large search trees, while focusing its effort on the most promising parts of the tree search (Browne et al., 2012). It has been shown to be an essential component for building an expert-level computer player for the game of Go (Silver et al., 2016), but it has also revealed to be efficient in other domains. In particular, planning or ordering is also a domain in which MCTS-based techniques are widely used (Asmuth and Littman, 2012; Balla and Fern, 2009; Bjarnason

et al., 2009). MCTS enjoys several advantages compared to traditional search algorithms. Notably, MCTS is:

- Asymmetric. MCTS expands the search tree asymmetrically, which is suitable for a large search space. It visits more frequently nodes that are likely to reach good goal states, and therefore focuses on exploring the promising parts of the tree.
- Anytime. MCTS can be terminated at any time and return the best solution found so far.

By combining Monte Carlo Tree Search with the ALS method, the *MCTS-ALS* algorithm is designed, aiming at finding a good ordering of LACs and getting an approximate circuit with higher quality improvement. The overall algorithm is shown in Algorithm 5 and some supporting functions are shown in Algorithm 6. The algorithm is based on the most popular version of MCTS, the *upper confidence bound for trees* (*UCT*) algorithm (Browne et al., 2012). Domain knowledge of ALS is integrated into it.

The algorithm first creates the root node R of the search tree, which corresponds to the original netlist C_{ori} (Line 1). Like any MCTS algorithm, it iteratively expands the search tree by repeating four basic steps—selection, expansion, playout, and backpropagation (Lines 4–7)—until a computational budget is reached. For simplicity, a total runtime limit T (Line 3) is used. Unlike standard MCTS, the best goal state (*i.e.*, approximate circuit) C_{ax} found so far (Line 9) is kept, which is returned at the end. Therefore, an approximate design can be obtained at any time, and its quality increases with more runtime.

Algorithm 5: The MCTS-ALS algorithm.

Input : the original netlist C_{ori}, the error rate threshold T_{er}, and the runtime limit T.
Output: an approximate netlist C_{ax}.

```
1  Create root node R with the original netlist C_ori;
2  Final approximate circuit C_ax ← C_ori;
3  while runtime is less than T do
4  |  V ← Selection();
5  |  New expanded node N_e ← Expansion(V);
6  |  (new approx. circuit C_new, reward Δ) ← Playout(N_e);
7  |  Backpropagation(N_e, Δ);
   |  // Area(C) is the area of the circuit C
8  |  if Area(C_ax) > Area(C_new) then
9  |  |  C_ax ← C_new;
10 return C_ax
```

An illustration of the four basic steps, which are described in detail next, is shown in Fig. 3.3. Conceptually, the selection step chooses a promising node V in the current search tree to further expand during the expansion step, which adds a new node N_e by applying a new LAC (if any) to V. The playout step applies a random sequence of LACs to N_e, which provides a noisy evaluation Δ to it. This information is backpropagated to N_e

and its predecessors in the backpropagation step. Some of these steps call a supporting function *GetValidLACs*, which returns a set of valid LACs for a circuit. It will be described in detail later.

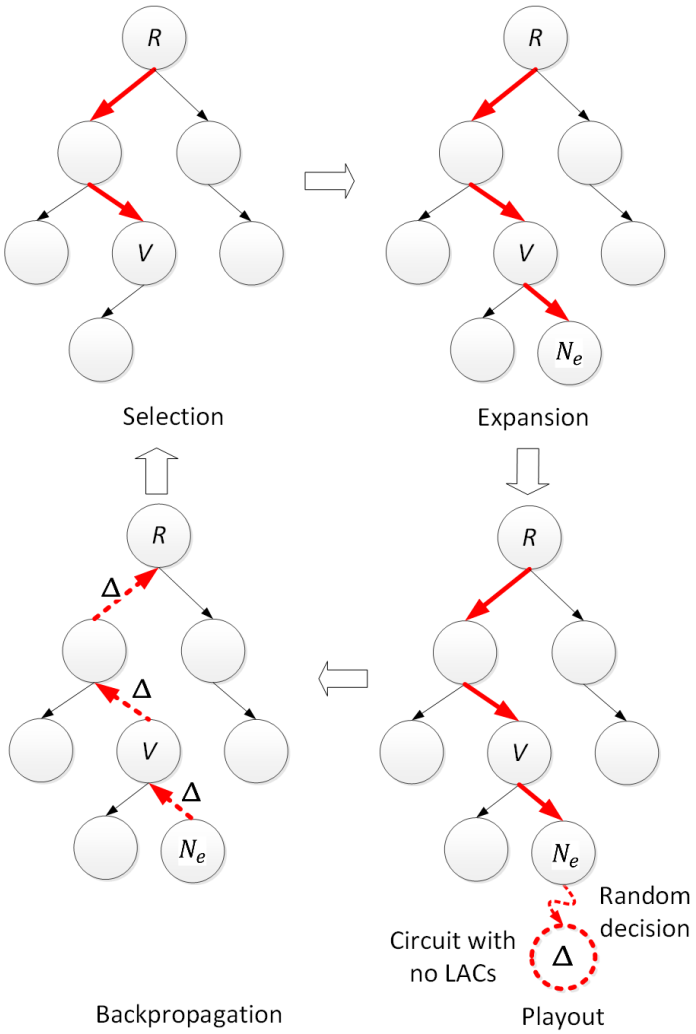


Figure 3.3: The steps of Monte Carlo tree search (MCTS).

Selection This step (Algorithm 6, Lines 1–7) traverses the current search tree to find the most promising node to expand. It starts from root R and iteratively selects child nodes until a candidate node V is reached. A *candidate node* is either a goal state or an expandable node, which is a node that still has some LACs that have not been tried yet.

For a non-candidate node V, the selection rule for the next child node V_j consists in maximizing the following expression (Algorithm 6, Line 5):

$$\frac{Q(V_j)}{N(V_j)} + \beta\sqrt{\frac{2\ln N(V)}{N(V_j)}}, \tag{3.1}$$

where $Q(V_j)$ is the sum of rewards Δ received in node V_j, $N(V_j)$ (resp. $N(V)$) counts how many times node V_j (resp. V) has been explored, and β is a constant parameter. The reward provides a noisy evaluation of the quality of a sequence of actions. In the general case, an MCTS algorithm aims at finding a sequence of actions that maximizes the reward. In the context of ALS, it is expected to choose a sequence of LACs to apply to maximize the final area saving of the approximate circuit over the original circuit. Thus, the reward Δ is defined as

$$\Delta = 1 - \frac{Area(C_{new})}{Area(C_{ori})}, \tag{3.2}$$

where $Area(C_{new})$ is the area of the new approximate circuit derived from playout (see the "playout" paragraph later for detail) and $Area(C_{ori})$ is the area of the original circuit. The reward defined in this form is also called *area reduction ratio* (ARR).

The values of Q and N are accumulated through the backpropagation step (see the "backpropagation" paragraph later for detail). Empirically, β is set to $\sqrt{2}$ when the reward Δ is limited within the range $[0, 1]$ (Kocsis et al., 2006), which is the case for the choice of Δ in this chapter.

Eq. (3.1) computes a high-probability upper confidence bound on the estimation of the true value of a node. It is analyzed in the reference (Kocsis and Szepesvári, 2006) as a selection rule in MCTS algorithms. It balances the exploitation of the action currently believed to be optimal with the exploration of other actions that may turn out to be superior in the long run (Browne et al., 2012). More specifically, the first term of Eq. (3.1), $Q(V_j)/N(V_j)$, gives an estimated expectation of the reward for choosing node V_j. A potential good ordering of LACs corresponds to a path from the root to a goal state in which the nodes have high expectations of rewards. Thus, the first term encourages the exploitation of promising LACs. However, when $N(V_j)$ is small, which means that node V_j has only been explored a few times, the estimation $Q(V_j)/N(V_j)$ may be far from the true expectation. Thus, the second term, which is larger if $N(V_j)$ is small, encourages the selection of under-explored nodes.

Expansion This step (Algorithm 6, Lines 8–14) is an essential operation to grow the search tree. If V is a goal state (*i.e.*, no valid LACs), V itself is returned. Otherwise, an unexplored valid LAC of candidate V is chosen and applied to generate a new approximate circuit, *i.e.*, a new node N_e is added to the current tree as a child of V. Finally, the node N_e is returned (Algorithm 6, Lines 11–14).

Playout This step (Algorithm 6, Lines 15–24) is a Monte Carlo process. It starts from the new node N_e and iteratively applies valid LACs at random to reach an approximate circuit with no more valid LACs, which

corresponds to a goal state (Algorithm 6, Lines 18–21). Since a sequence of introduced LACs may induce some redundancies in the circuit, the final circuit generated by playout is optimized by a traditional logic synthesis tool (Algorithm 6, Line 22). This could further reduce the area, while not increasing the ER. Finally, the final approximate circuit and the reward Δ calculated by Eq. (3.2) are returned.

Backpropagation This step (Algorithm 6, Lines 25–29) updates two pieces of vital information in each node along the path from the newly expanded node N_e back to root R. They are the total reward Q and the total visited time N. For each visited node, total reward Q increases by a reward Δ returned from the playout step and total visited time N is incremented by 1. These two values are used in Eq. (3.1) to guide the selection step in future rounds.

Algorithm 6: The supporting functions used in the MCTS-ALS algorithm.

```
   // Ckt(V) is the circuit of node V
 1 Function Selection():
 2     Node V ← root node R;
 3     Candidate LAC set S_LAC ← GetValidLACs(Ckt(V), T_er);
 4     while S_LAC ≠ ∅ and all LACs in S_LAC have been explored do
 5         V ← arg max_{V_j ∈ V's children} Q(V_j)/N(V_j) + β·sqrt(2 ln N(V) / N(V_j));
 6         S_LAC ← GetValidLACs(Ckt(V), T_er);
 7     return V
 8 Function Expansion(V):
 9     S_LAC ← GetValidLACs(Ckt(V), T_er);
10     if S_LAC = ∅ then return V ;
11     Randomly choose one unexplored LAC ∈ S_LAC;
12     Add a new child N_e to V;
13     Apply LAC on Ckt(V) and get Ckt(N_e);
14     return N_e
15 Function Playout(N_e):
16     C_new ← Ckt(N_e);
17     S_LAC ← GetValidLACs(C_new, T_er);
18     while S_LAC ≠ ∅ do
19         Choose LAC ∈ S_LAC randomly;
20         Simplify C_new with LAC;
21         S_LAC ← GetValidLACs(C_new, T_er);
22     Simplify C_new with a traditional logic synthesis tool;
23     Reward Δ ← 1 − Area(C_new) / Area(C_ori);
24     return (C_new, Δ)
25 Function Backpropagation(V, Δ):
26     while V is not null do
27         Q(V) ← Q(V) + Δ;
28         N(V) ← N(V) + 1;
29         V ← parent of V;
```

Speed-up by Limiting the Number of Valid LACs The selection, expansion, and playout steps all call function *GetValidLACs* that returns a set of valid LACs to consider for a circuit under an ER threshold. However, the number of valid LACs is quadratic to the number of signals in the circuit. If all valid LACs for each node are considered, then the tree would grow too fast, which would lead to a prohibitive amount of computation. Thus, to make the algorithm practical, it is proposed to only select a subset of valid LACs: after obtaining all valid LACs, the algorithm only keeps the top B of them that introduce the smallest ERs to the circuit as the LAC candidates. This technique can reduce the number of unpromising LACs and hence, avoid the unnecessary exploration of unpromising tree branches with higher ERs.

3.6 Experimental Results

This section presents the experimental results of the proposed methods based on advanced search algorithms. The algorithms are implemented in C++ and tested on a desktop computer with an eight-core 3.6GHz Xeon CPU, operating on Ubuntu 16.04.

The SASIMI LAC mentioned in Section 3.2 is applied to simplify the circuit. ER is the selected error metric, which is measured by performing logic simulation. Assume that all primary input vectors are uniformly distributed. For each algorithm, 100,000 input vectors are generated at the beginning, and they are applied in each logic simulation for measuring the ER throughout the entire process of approximate circuit generation. However, when measuring the ER of the final approximate circuit, 1,000,000 different input vectors are used to measure the ER more accurately. Note that due to the randomness, if the final measured ER exceeds the threshold, it is proposed to undo the last applied LAC and repeat this process, until a circuit with the final measured ER smaller than the threshold is reached. The ARR, defined as the ratio of the reduced area over the original area (see Eq. (3.2)), is used to evaluate the quality of inexact designs. Although circuit delay is not the focus here, all produced approximate designs do not have delay increase over their original designs thanks to proper choices of LACs.

The ISCAS85 benchmark suite is used, and those benchmarks are mapped with MCNC generic standard cell library (Yang, 1991). The delay, area, and number of primary inputs/outputs of each benchmark are listed in Table 3.1. The traditional logic synthesis and technology mapping are performed by the logic synthesis tool ABC (Mishchenko et al., 2022).

Table 3.1: Benchmark information

Circuit	Area	Delay	# PIs/POs
C432	309	21.9	36/7
C499	792	15.3	41/32
C880	629	16.4	60/26
C1908	747	24	33/25
C2670	1374	15.7	233/140
C3540	1915	28.7	50/22
C5315	2408	30.2	178/123
C7552	3328	25.2	207/108

3.6.1 Comparison of Approximate Circuit Quality for Different Methods

The proposed approaches, BS-ALS and MCTS-ALS, are compared with the basic greedy method. The greedy method used for comparison here is a state-of-the-art method described in the reference (Su et al., 2018). It scores a LAC by the ratio of the area reduction over the increased ER. In the BS-ALS algorithm, $K = 10$ best candidates are kept for each level of the search tree. In the MCTS-ALS algorithm, the top $B = 100$ LACs with lower ERs are kept.

Table 3.2: Area reduction ratios (ARRs) of three different ALS methods.

Circuit	Method	C432	C499	C880	C1908	C2670	C3540	C5315	C7552	Average
ARR at $ER = 0.5\%$	Greedy	**18.12%**	**1.14%**	**4.29%**	3.21%	27.87%	3.29%	2.41%	13.64%	9.25%
	BS	**18.12%**	**1.14%**	**4.29%**	3.48%	28.31%	3.29%	**3.57%**	15.20%	9.68%
	MCTS	**18.12%**	**1.14%**	**4.29%**	**6.69%**	**31.73%**	**6.63%**	**3.57%**	**22.63%**	**11.85%**
ARR at $ER = 1\%$	Greedy	18.12%	**2.27%**	4.61%	7.63%	28.82%	3.92%	3.53%	14.63%	10.44%
	BS	18.77%	**2.27%**	4.61%	7.90%	30.71%	3.97%	**4.98%**	15.32%	11.07%
	MCTS	**19.09%**	**2.27%**	**5.25%**	**12.58%**	**33.70%**	**10.86%**	**4.98%**	**22.63%**	**13.92%**
ARR at $ER = 3\%$	Greedy	18.12%	9.09%	6.68%	40.96%	32.53%	8.36%	4.44%	15.75%	16.99%
	BS	21.36%	8.84%	6.84%	42.97%	32.61%	8.30%	4.98%	16.56%	17.81%
	MCTS	**24.92%**	**9.72%**	**17.01%**	**50.60%**	**38.57%**	**15.87%**	**5.11%**	**23.74%**	**23.19%**
ARR at $ER = 5\%$	Greedy	20.06%	14.27%	18.28%	61.98%	40.90%	14.10%	5.15%	16.26%	23.88%
	BS	**33.01%**	13.76%	18.28%	**62.25%**	40.03%	14.20%	5.15%	17.67%	25.54%
	MCTS	**33.01%**	**19.95%**	**19.71%**	**62.25%**	**41.85%**	**16.61%**	**5.56%**	**25.06%**	**28.00%**
Average ARR	Greedy	18.61%	6.69%	8.47%	28.45%	32.53%	7.42%	3.88%	15.07%	15.14%
	BS	22.82%	6.50%	8.51%	29.51%	32.91%	7.44%	4.67%	16.19%	16.02%
	MCTS	**23.79%**	**8.27%**	**11.57%**	**33.03%**	**36.46%**	**12.49%**	**4.81%**	**23.51%**	**19.24%**

The results of the three methods are shown in Table 3.2. ARRs for 4 ER thresholds (0.5%, 1%, 3%, 5%) are reported for each benchmark. “Greedy”, “BS”, and “MCTS” in the table correspond to the greedy, BS-ALS, and MCTS-ALS methods, respectively. The best quality improvement at a certain ER constraint for a benchmark is highlighted in bold.

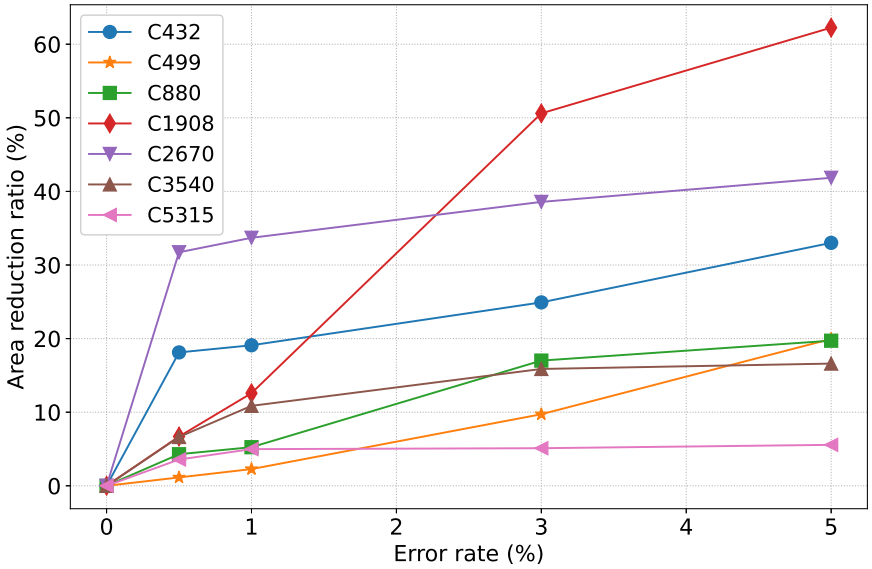


Figure 3.4: Area reduction ratio v.s. error rate for MCTS-ALS.

Note that for most cases, BS-ALS reduces more area than the greedy method. On average, BS-ALS behaves slightly better and improves the ARR by 0.43%, 0.63%, 0.82%, and 1.66% for the four ER thresholds over the greedy method. Furthermore, MCTS can find even better orderings of LACs and further improve the quality of approximate circuits. Indeed, MCTS-ALS performs best for all cases in the table, as all the values in the "MCTS" columns are highlighted in bold. On average, it improves ARR by 2.6%, 3.48%, 6.2%, and 4.12% under the four ER thresholds over the greedy method. In particular, MCTS-ALS further reduces the area by 12.95% over the greedy method for circuit C432 when $ER = 5\%$. Fig. 3.4 plots the relationship between ER and ARR for all the benchmarks using MCTS-ALS. It can be seen that it could reduce 15%–60% area for most benchmarks under 5% ER threshold.

It deserves a mention that BS-ALS sometimes generates worse approximate designs than the greedy method (e.g., circuit C499 under ER threshold of 5%). Since beam search is an extension of the basic greedy search, it also cannot guarantee finding the best solution in the search space due to falling into a local minimum. Even if beam search keeps more than one most promising state, it is possible to make bad choices at some steps, resulting in missing an optimal goal state eventually. That is why BS-ALS sometimes does not beat the basic greedy method.

3.6.2 Comparison of Runtime for Different Methods

With regard to the runtime to obtain the results in Table 3.2, the greedy method takes 16 minutes on average, while the BS-ALS method consumes about 10× time compared with the greedy method. This is reasonable since in the experiments, $K = 10$ parallel "search threads" are maintained. For the MCTS-ALS method, the given runtime limit T is set as 24 hours and it generates an approximate circuit as good as possible.

Since the runtime mentioned above is not equivalent for different ordering methods, the tradeoff between runtime and ARR for different ALS methods is explored. To illustrate how much time the MCTS-ALS method consumes to reach the same approximate circuit quality as the greedy method, MCTS-ALS flow is terminated as soon as it finds an approximate design with ARR larger than or equal to that of the greedy flow. The terminating time T_M is recorded and compared with the time T_G of the greedy method in Table 3.3. The ratio $\frac{T_M}{T_G}$ is also listed in the table. Three different ERs of 1%, 3%, and 5% are considered. On average, in order to get the same quality, the MCTS-ALS method requires 4× runtime compared to the greedy method. Although the runtime of MCTS is longer than the greedy method, it has the advantage of continuously improving the quality as runtime increases, which will be discussed in the following section. It is beneficial when quality is the primary concern.

Table 3.3: Runtime of the greedy and the MCTS-ALS methods for reaching the same area improvement. T_G and T_M denote the time in seconds of the greedy and the MCTS-ALS methods, respectively.

Circuit	$ER = 1\%$			$ER = 3\%$			$ER = 5\%$			Average
	T_G	T_M	T_M/T_G	T_G	T_M	T_M/T_G	T_G	T_M	T_M/T_G	T_M/T_G
C432	9	10	1.1	15	120	8.0	17	101	5.9	5.0
C499	6	11	1.8	27	53	2.0	36	35	1.0	1.6
C880	17	18	1.1	21	21	1.0	32	226	7.0	3.0
C1908	76	151	2.0	211	639	3.0	213	998	4.7	3.2
C2670	266	264	1.0	354	368	1.0	438	4326	9.9	4.0
C3540	469	3802	8.1	699	3492	5.0	1513	1510	1.0	4.7
C5315	971	2900	3.0	1329	1300	1.0	2117	4266	2.0	2.0
C7552	2821	28000	9.9	2999	18993	6.3	3101	28912	9.3	8.5
Average	579.4	4394.5	3.5	706.9	3121.5	3.4	933.4	5387.5	8.2	4.0

3.6.3 Quality Configurable ALS Flow with MCTS

MCTS can be stopped at any time and return the best ordering of LACs, which corresponds to an approximate circuit with the highest quality (such as ARR) so far. The quality of an approximate circuit gradually improves as iteration (*i.e.*, a loop of selection, expansion, playout, and backpropagation) number increases. Fig. 3.5 shows the relationship between ARR and iteration time of MCTS using circuit C1908 and ER threshold of 3% as an

example. The staircase-like curve in blue plots ARRs versus iteration times for MCTS-ALS, while the horizontal line in orange, drawn as a reference, is the ARR of the approximate circuit generated by the greedy method. It can be seen that the quality produced by MCTS-ALS grows in a staircase-like way, since the current best area is updated at the end of each loop, as shown in Algorithm 5. These two curves intersect at the *third* iteration of the MCTS-ALS method. This means after three loops of MCTS, it has already found a LAC ordering that produces an approximate circuit with the same quality (*i.e.*, 40.9% area reduction) as the greedy method.

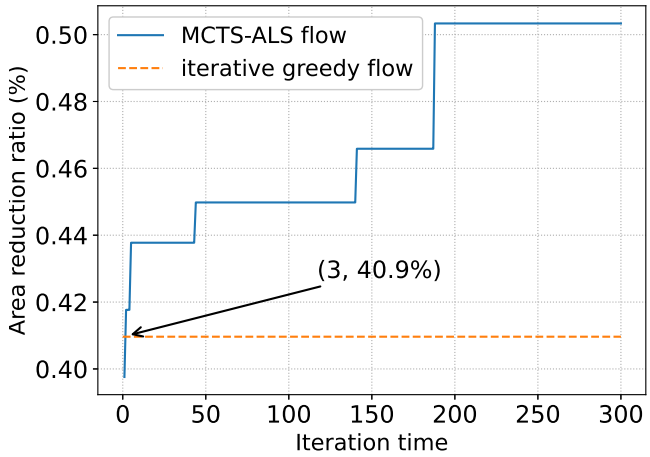


Figure 3.5: Circuit quality improves with iteration number for the MCTS-ALS method. The results are obtained on circuit C1908 under ER threshold of 3%.

Consequently, the MCTS-ALS method can be flexibly used to generate inexact designs with different quality in accordance with the user requirement. In case of a strict demand on quality improvement from the user, MCTS-ALS could be given more computational budgets to produce a better approximate design. In contrast, if the user does not care much about the quality, MCTS-ALS could produce a satisfying design in a shorter time.

3.7 Summary

In this chapter, advanced search methods are proposed to determine a good order of applying the LACs for multi-level approximate logic synthesis. Two non-greedy ALS flows, BS-ALS and MCTS-ALS, are proposed based on beam search and MCTS, respectively. They show improvement over the basic greedy search method. The experimental results showed that MCTS-ALS performs best. It has the attractive feature of continuously improving the design quality as more runtime is allowed. As a future work, it will be considered to further speed up the MCTS-ALS algorithm.

Chapter 4

HEDALS: Highly Efficient Delay-Driven Approximate Logic Synthesis Flow

This chapter presents a novel delay-driven ALS flow called HEDALS that can generate approximate circuits with dramatically reduced delays very efficiently.

4.1 Motivations and Overview

As mentioned in Section 1.4.1, although significant progress has been made in ALS in recent years, most ALS methods mainly focused on reducing the circuit area, such as the works in the references (Shin and Gupta, 2011; Venkataramani et al., 2013; Wu and Qian, 2016; Liu and Zhang, 2017; Meng et al., 2020; Witschen et al., 2022; Ma et al., 2021). Delay reduction is usually a byproduct of these ALS methods, and the potential of ALS in optimizing delay has not been fully explored. Meanwhile, many error-tolerant applications have stringent timing constraints, such as real-time signal processing. For them, delay, instead of area, is the primary concern. Therefore, a delay-oriented ALS flow is preferred for these applications.

Given the difficulty in optimizing approximate circuits globally, existing ALS methods usually repeatedly apply LACs to the nodes in a circuit, such as replacing nodes by constant 0s or 1s, until the given error constraint is reached. In these methods, where area is the primary optimization target, all nodes in a circuit are treated as candidates for approximation, since they contribute to the area equally regardless of their locations. However, it is not the case for delay optimization. Circuit delay is determined by the critical paths. Thus, instead of considering all nodes in a circuit, effort should be put into the nodes on the critical paths. Furthermore, even

approximating some nodes on the critical paths may be ineffective, as there may exist multiple critical paths with the same delay. If the approximation only reduces the delays of some, but not all, critical paths, the overall circuit delay remains unchanged. Instead, a better choice to reduce the circuit delay is to shorten all the critical paths simultaneously.

To address the above issues, this chapter proposes *HEDALS*, a highly efficient delay-driven approximate logic synthesis framework. To effectively reduce the delay, it only focuses on the nodes on the critical paths and only considers the LACs applied on these nodes. Moreover, it selects an optimized set of LACs and applies them simultaneously to the circuit so that all the critical paths are shortened, while the induced error is minimized.

The main contributions of this chapter are as follows:

- To facilitate the finding of an optimized set of LACs that can reduce the circuit delay, while minimizing the induced error, this chapter proposes a novel data structure called *critical error graph* (*CEG*).
- This chapter proposes a maximum flow-based method to find an optimized LAC set with the help of CEG and an efficient model to estimate the error caused by a set of LACs.
- This chapter proposes a priority cut-based method to further improve the LAC set found by the maximum flow-based method.
- Based on the above techniques, this chapter proposes HEDALS, a delay-oriented ALS framework supporting various LAC types, circuit representations, and average error metrics.

The experimental results show that HEDALS can rapidly synthesize approximate circuits with significantly reduced delay. Besides, it has good scalability and wide applicability in terms of the supported LAC types, circuit representations, and error metrics. Compared to a state-of-the-art method, on average, HEDALS can reduce the circuit delay by 32.3%, while being 167× faster.

The rest of the chapter is organized as follows. Section 4.2 introduces the background. Sections 4.3–4.5 first introduce the overall HEDALS framework and then present its details. Section 4.6 shows the experimental results. Finally, Section 4.7 summarizes the chapter.

4.2 Preliminaries

This section introduces the preliminaries of HEDALS. For the circuit terminologies and average error metrics, please refer to Section 2.2. Next, circuit timing will be introduced as follows.

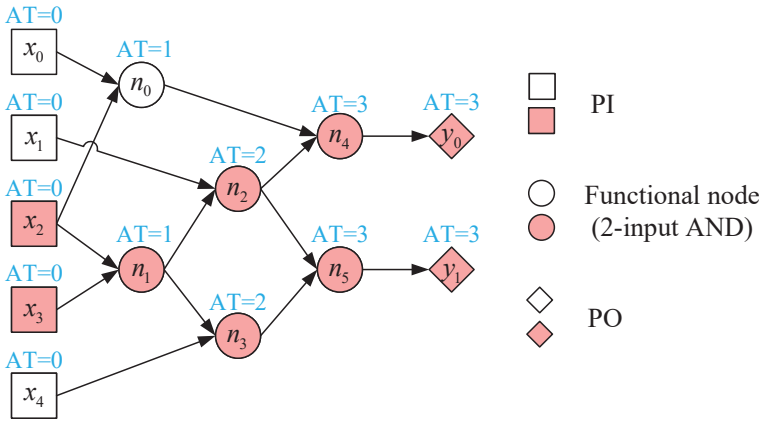


Figure 4.1: An example circuit in the AIG representation, where all the edges are non-complemented. The ATs of the nodes are marked in blue. The red nodes are on the critical paths.

In this chapter, when calculating delays, only node delays are considered, and the wire delays are ignored. The method of calculating the node delay varies for different circuit representations. In AIGs, MIGs, and LUT networks, the node delay is set as 1. In gate netlists, where each node is a gate from a standard cell library, the delay of a node can be looked up from the library according to its input transition time and output capacitance. The *arrival time* (AT) of a node is the largest delay of all paths from a PI to the node. A path from a PI to a PO with the maximum delay is a *critical path*, and the maximum delay is the *circuit delay*. Note that a circuit can have multiple critical paths. For example, for the AIG in Fig. 4.1, the AT of each node is shown in blue, and the red nodes are on the critical paths. This AIG has more than one critical path. The ATs, circuit delay, and critical paths can be obtained by *static timing analysis* (STA) (Sapatnekar, 2004).

4.3 Overall Framework of HEDALS

Given an exact circuit, HEDALS aims at synthesizing an approximate circuit with reduced delay without increasing its area. Notably, the circuit can be represented in a commonly-used form such as AIG, MIG, LUT network, and gate netlist. The error constraint can be any average error constraint, such as ER, MED, and MHD.

The overall flow of HEDALS is shown in Algorithm 7. Its inputs are an exact circuit G and an error upper bound e_b. It outputs an approximate circuit G_{apx} with an error not exceeding e_b. Line 1 initializes the approximate circuit G_{apx} as the exact circuit G and a Boolean variable *HasSol* as true. Then, Lines 2–3 iteratively simplify the approximate circuit G_{apx}. Finally, Line 4 applies traditional delay-oriented logic synthesis and technology mapping to produce a mapped approximate circuit G_{apx}, which is then returned (Line 5).

Algorithm 7: The procedure of HEDALS.

Input: an exact circuit G and an error upper bound e_b.
Output: an approximate circuit G_{apx} with an error $\leq e_b$.

```
1 G_apx ⇐ G; HasSol ⇐ true;
2 while HasSol do
      // can further reduce delay
3     (HasSol, G_apx) ⇐ FindApplyOptLACSet(G_apx, e_b);
4 G_apx ⇐ SynthesizeAndMap(G_apx);
5 return G_apx
```

In each iteration, the function *FindApplyOptLACSet* is called to do the simplification (Line 3). Its main task is to find a set of LACs, marked as $\mathcal{L}$, and apply them to the approximate circuit G_{apx} so that the circuit delay decreases by some amount. For a LAC set $\mathcal{C}$, *Error*($\mathcal{C}$) is used to denote the error of the approximate circuit obtained by applying the LACs in set $\mathcal{C}$. It is also referred to as the *error of the LAC set* $\mathcal{C}$. Since the LAC set $\mathcal{L}$ found in each iteration reduces the circuit delay by some amount, to minimize the delay of the final approximate circuit, a good heuristic is to maximize the number of iterations. To achieve this, this work aims at minimizing the error of the LAC set $\mathcal{L}$ in each iteration. Thus, a problem is formulated as follows:

Problem 4.1. Given the set $\mathbb{L}$ of all possible LAC sets in an approximate circuit G_{apx}, find the LAC set $\mathcal{L} \in \mathbb{L}$ with the minimum error, while also satisfies that

1. after applying all LACs in $\mathcal{L}$, the circuit delay is reduced;

2. *Error*($\mathcal{L}$) is no more than the error bound e_b.

Note that the solution space of Problem 4.1 is very large, since it includes all possible LAC sets in the approximate circuit G_{apx}. On the one hand, there are many sets of nodes in a circuit, on which LACs can be identified. On the other hand, even for a certain node set, there are many ways of introducing LACs to the nodes in the set, since each node in the set generally has multiple LACs.

The function *FindApplyOptLACSet* provides an efficient solution to Problem 4.1. To reduce the solution space, it exploits a novel data structure, *critical error graph* (*CEG*), which will be elaborated in Section 4.4. Using the CEG, two implementations of *FindApplyOptLACSet* are proposed to solve Problem 4.1, *i.e.*, a maximum flow-based method and a priority cut-based method. Their details will be described in Section 4.5. Both methods first pre-process the current approximate circuit and then find an optimized LAC set from the pre-processed circuit. If an optimized set $\mathcal{L}$ is found, then the function *FindApplyOptLACSet* sets the variable *HasSol* to true, applies all LACs in $\mathcal{L}$ to simplify the pre-processed circuit, and returns the resulting circuit (Line 3). However, it is also possible that for a pre-processed circuit, no LAC set satisfying the conditions

in Problem 4.1 can be found. Then, the function *FindApplyOptLACSet* sets *HasSol* to false and returns the pre-processed circuit. In this case, the loop is terminated.

4.4 Critical Error Graph

This section presents the details of CEG, which is a key data structure used in the function *FindApplyOptLAC-Set*. Since the solution to Problem 4.1 should be a LAC set that can reduce the circuit delay, this section first analyzes which LAC sets can achieve this in Section 4.4.1. However, as the number of LAC sets that can reduce delay is large, it is proposed to focus on a promising subset of these LAC sets in Section 4.4.2, which naturally leads to CEG. This section finally concludes how CEG is built in Section 4.4.3.

4.4.1 LAC Sets for Delay Reduction

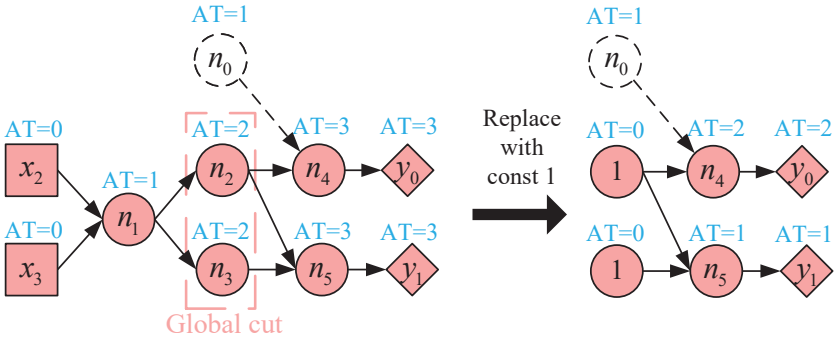


Figure 4.2: An example of shortening all the critical paths in the critical graph by applying delay-reducing LACs to the nodes in a global cut of the critical graph. The left figure is the critical graph of the AIG in Fig. 4.1.

Clearly, in order to effectively reduce the delay, only the nodes on the critical paths need to be focused. They form a *critical graph* formally defined below.

Definition 4.1. The **critical graph** $CG = (V, E)$ for a circuit is a subgraph of it, where V and E are the sets of nodes and edges, respectively, on the critical paths of the circuit.

If a node in the critical graph corresponds to a PI/PO/functional node in the original circuit, it is also called a PI/PO/functional node in the critical graph.

Example 4.1. The left part of Fig. 4.2 shows the critical graph for the AIG in Fig. 4.1, where the set of nodes $V = \{x_2, x_3, n_1, n_2, n_3, n_4, n_5, y_0, y_1\}$ forms the critical paths of the AIG. In the critical graph, the PIs are x_2

and x_3, the POs are y_0 and y_1, and the functional nodes are $n_1, \ldots, n_5$. Note that the node n_0 does not belong to the critical graph; it is shown to illustrate the computation of AT clearly.

Because there usually exist multiple critical paths in a circuit, thus, to reduce the circuit delay, it is not enough to shorten one critical path. Instead, all the critical paths should be shortened. A *global cut* gives us a way to achieve this.

Definition 4.2. A **global cut** of the critical graph is a set of nodes satisfying: 1) each node in the set is a functional node, *i.e.*, neither a PI nor a PO, and 2) each path from a PI to a PO of the critical graph passes at least one node in the set.

For instance, $\{n_2, n_3\}$ is a global cut of the critical graph shown in the left part of Fig. 4.2. If for each node n in a global cut, a LAC on node n is applied such that n's AT decreases, then all the critical paths are shortened, and the circuit delay is reduced. Such a LAC on a node n leading to a reduction of n's AT is called a *delay-reducing LAC* of node n.

Example 4.2. Suppose that the constant LAC (Shin and Gupta, 2011) is applied to the circuit shown in the left part of Fig. 4.2. By replacing each node in the global cut $\{n_2, n_3\}$ with a constant 1, the ATs of n_2 and n_3 are both reduced to 0. Even without further applying constant propagation, the AT of the PO y_0 is reduced from 3 to 2, and that of the PO y_1 is reduced from 3 to 1. Hence, the circuit delay is reduced to 2.

The above example shows that constant LAC is delay-reducing. Besides it, most LACs proposed in previous works (Venkataramani et al., 2013; Wu and Qian, 2016; Liu and Zhang, 2017; Meng et al., 2020; Witschen et al., 2022) are also delay-reducing and hence, are applicable to HEDALS.

By the above analysis, it can be concluded that a set of delay-reducing LACs applied to the nodes on a global cut of the critical graph of the circuit can reduce the circuit delay. Such a LAC set is called a *delay-reducing LAC set*.

4.4.2 Promising Subset of All Delay-Reducing LAC Sets and Critical Error Graph

The optimal solution to Problem 4.1 must be a delay-reducing LAC set. However, the number of delay-reducing LAC sets is usually very large. First, for a single global cut, its number of delay-reducing LAC sets grows exponentially with the size of the cut, as each node in the cut may have multiple delay-reducing LACs. Moreover, for a large circuit, the number of global cuts of its critical graph is usually large. Thus, the number of delay-reducing LAC sets is very large, and it is prohibitive to evaluate all of them.

To reduce the complexity, a feasible way is identifying a promising subset of all delay-reducing LAC sets with small errors. Note that the error metric can be any average error metric of interest, such as ER, MED, and MHD, which can be calculated as shown in Section 2.2.2. To achieve the above purpose, for each node n in the critical graph, after obtaining the errors of all of its delay-reducing LACs, only the one with the minimum error is kept. It is called the *min-error LAC* of node n. Then, for each global cut $\mathcal{N}$ of the critical graph, this work only maintains a single delay-reducing LAC set that consists of the min-error LAC for each node on the cut. It is called the *leading LAC set* of cut $\mathcal{N}$. The leading LAC sets of all the global cuts of the critical graph form the promising subset. By focusing on the leading LAC sets, Problem 4.1 can be converted into the following one.

Problem 4.2. Given an approximate circuit G_{apx}, among all the leading LAC sets of all the global cuts of the critical graph of G_{apx}, find one with the minimum error, while also satisfying that the error does not exceed the error bound e_b.

Since the leading LAC set of any global cut is a delay-reducing LAC set, the circuit delay is guaranteed to be reduced after applying all LACs in the final solution set. Thus, the delay constraint in Problem 4.1 does not appear in Problem 4.2.

An essential component of Problem 4.2 is the critical graph of the circuit with each node in the graph associated with its min-error LAC. This leads to the following definition of the critical error graph (CEG) of a circuit. Specifically, suppose that before applying any LAC, the error of the current approximate circuit, which is called the *base error*, is e_{base}. After applying the min-error LAC of node n, the error of the resulting approximate circuit is e. The increased error caused by the LAC is $\Delta e = e - e_{base}$, which is also called the *minimum error increase (MEI)* of node n. The *CEG* of the approximate circuit is the critical graph of the circuit with each functional node associated with its MEI. For example, Fig. 4.3 shows the CEG of the AIG in Fig. 4.1, where the MEI of each functional node is put near it. CEG facilitates the solving of Problem 4.2, which will be described in Section 4.5.

4.4.3 Building Critical Error Graph

Algorithm 8 shows how the CEG of an approximate circuit is built. Initially, Line 1 builds the critical graph *CG* from the nodes and the edges on the critical paths of the circuit G_{apx}. Then, for each functional node n in *CG*, Line 3 generates a set $\mathcal{U}$ of delay-reducing LACs of n. After that, it obtains the error of applying each LAC $l \in \mathcal{U}$ to the circuit G_{apx} (Lines 4–5), which is achieved by an efficient error estimation method, VECBEE (Su et al., 2022). Line 6 then finds the min-error LAC of node n, followed by Line 7, which obtains the MEI of

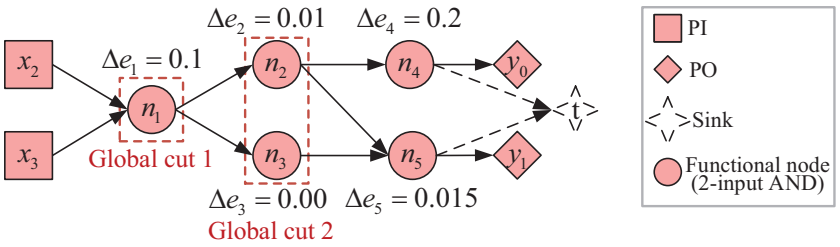


Figure 4.3: The critical error graph (CEG) of the AIG in Fig. 4.1. The value Δe_i beside the functional node n_i represents the minimum error increase (MEI) of n_i, where the error metric can be any average error metric of interest, such as ER, MED, and MHD. Note that the sink node t does not belong to the CEG; it is only used for obtaining a set of global cuts in Section 4.5.2.

Algorithm 8: *BuildCriticalErrorGraph*(G_{apx}), a function for building the critical error graph of a circuit.

Input: an approximate circuit G_{apx}.
Output: the critical error graph of G_{apx}.
1 Build critical graph CG based on the critical paths in G_{apx};
2 **foreach** *functional node* n *in graph* CG **do**
3 Generate a set $\mathcal{U}$ of delay-reducing LACs of n;
4 **foreach** $LAC\ l \in \mathcal{U}$ **do**
5 Obtain the error of applying LAC l to G_{apx};
6 $n.MinErrorLAC \Leftarrow$ the LAC of n with the smallest error;
7 $n.MEI \Leftarrow$ increased error caused by applying $n.MinErrorLAC$;
8 **return** CG

node n. Finally, Line 8 returns the critical graph CG with each functional node associated with its MEI, *i.e.*, the CEG.

4.5 Obtaining an Optimized LAC Set

This section presents the details of the function *FindApplyOptLACSet*, which provides a solution to Problem 4.2. For a large circuit with many global cuts of its critical graph, it is still very challenging to exactly solve Problem 4.2 to find a leading LAC set with the minimum error. Instead, two methods are presented to obtain an optimized leading LAC set with a low error. The first is a maximum flow-based method. The second is a priority cut based-method, which builds upon and improves the maximum flow-based method.

4.5.1 Maximum Flow-Based Method

A solution to Problem 4.2 usually needs to obtain the error of a LAC set, which is typically done by time-consuming logic simulation. To improve the efficiency, the maximum flow-based method relies on an efficient linear model to estimate the error of a LAC set. This section first introduces the model and then presents the

method using the model.

Estimation of Error of a LAC Set

Consider the leading LAC set $\mathcal{L} = \{l_1, l_2, \ldots, l_m\}$ of a global cut with m nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_m\}$, where l_i $(1 \leq i \leq m)$ is the min-error LAC of node n_i. Assume that before the LACs are applied, the circuit has a base error, e_{base}, and that after applying the LAC l_i alone, the resulting approximate circuit has an error of e_i. By definition, the MEI of node n_i is $\Delta e_i = e_i - e_{base}$. In a previous work in the reference (Zhou et al., 2018), the error of the leading LAC set $\mathcal{L}$ is estimated by a linear model as follows:

$$Error(\mathcal{L}) \approx e_1 + e_2 + \cdots + e_m. \tag{4.1}$$

Since $\Delta e_i = e_i - e_{base}$, we also have

$$\begin{aligned} Error(\mathcal{L}) &\approx (e_{base} + \Delta e_1) + \cdots + (e_{base} + \Delta e_m) \\ &= m e_{base} + \Delta e_1 + \Delta e_2 + \cdots + \Delta e_m. \end{aligned} \tag{4.2}$$

Clearly, the model accumulates the base error m times, which is unreasonable. In this work, it is improved as follows:

$$Error(\mathcal{L}) \approx e_{base} + \Delta e_1 + \Delta e_2 + \cdots + \Delta e_m. \tag{4.3}$$

The new model is more accurate, as the example below shows.

Example 4.3. In the critical graph shown in Fig. 4.3, consider two global cuts $\mathcal{N}_1 = \{n_1\}$ and $\mathcal{N}_2 = \{n_2, n_3\}$. Assume that the LACs l_1, l_2, and l_3 are the min-error LACs of the nodes n_1, n_2, and n_3, respectively. Then, the leading LAC sets of $\mathcal{N}_1$ and $\mathcal{N}_2$ are $\mathcal{L}_1 = \{l_1\}$ and $\mathcal{L}_2 = \{l_2, l_3\}$, respectively. Assume that the base error e_{base} is 0.5, and that the MEIs of n_1, n_2, and n_3 are $\Delta e_1 = 0.1$, $\Delta e_2 = 0.01$, and $\Delta e_3 = 0$, respectively. Note that LAC l_3 is an exact local change introducing no error.

By Eq. (4.2), the errors of $\mathcal{L}_1$ and $\mathcal{L}_2$ calculated by the previous model are $Error_{old}(\mathcal{L}_1) = 0.5 + 0.1 = 0.6$ and $Error_{old}(\mathcal{L}_2) = 2 \times 0.5 + 0.01 + 0 = 1.01$, respectively, which indicates that the LAC set $\mathcal{L}_1$ is better due to its smaller estimated error. However, since LAC l_3 is an exact local change, the error of $\mathcal{L}_2 = \{l_2, l_3\}$ equals that caused by applying l_2 only. Thus, the actual error of $\mathcal{L}_2$ is $Error_{actual}(\mathcal{L}_2) = 0.5 + 0.01 = 0.51$. Besides, the actual error of $\mathcal{L}_1 = \{l_1\}$ is $Error_{actual}(\mathcal{L}_1) = 0.5 + 0.1 = 0.6$. Therefore, the LAC set $\mathcal{L}_2$ is better in reality.

Now, applying the proposed model, *i.e.*, Eq. (4.3), we can obtain the errors of $\mathcal{L}_1$ and $\mathcal{L}_2$ as $Error_{new}(\mathcal{L}_1) = 0.5 + 0.1 = 0.6$ and $Error_{new}(\mathcal{L}_2) = 0.5 + 0.01 + 0 = 0.51$, respectively. Thus, $\mathcal{L}_2$ is better, which agrees with

the real situation. Furthermore, for this example, the errors estimated by the proposed model equal the actual values.

It should be noted that the proposed estimation may not be accurate, since the exact error increase of applying multiple LACs together may not equal the sum of the error increases of applying each individual LAC alone. Nevertheless, for the average error metrics such as ER, MED, and MHD, the sum is still a good first-order approximation and enables the design of a more efficient algorithm.

Entire Flow of Maximum Flow-Based Method

For simplicity, the sum of the MEIs of all nodes in a global cut is called *the MEI sum* of the global cut. By the proposed error estimation model shown in Eq. (4.3), the error of the leading LAC set of a global cut equals the MEI sum of the global cut plus the base error. Since the base error is the same for all the leading LAC sets, Problem 4.2 is converted into selecting the optimal global cut from the CEG with the minimum MEI sum.

To find the optimal global cut, the proposed method first maps the original CEG into a dual graph and then solves a network flow problem on it. The dual graph is built as follows.

(a) For each functional node n with MEI Δe in the CEG, A pair of nodes n_a and n_b and an edge from n_a to n_b with a capacity of Δe are added to the dual graph.

(b) For each edge from a functional node u to a functional node v in the CEG, an edge from u_b to v_a with an infinite capacity is added to the dual graph.

(c) A source node s is added. For each edge from a PI node to a functional node n in the CEG, an edge from s to n_a with an infinite capacity is added to the dual graph.

(d) A sink node t is added. For each PO node in the CEG, let its only fanin functional node be n. An edge from n_b to t with an infinite capacity is added to the dual graph.

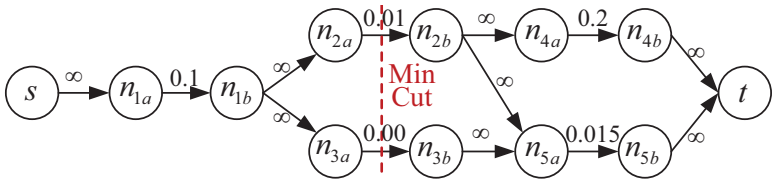


Figure 4.4: The dual graph built from the CEG shown in Fig. 4.3.

The dual graph built from the CEG shown in Fig. 4.3 is given in Fig. 4.4. The dual graph is a classic flow network (Cormen et al., 2009). For a flow network, a *cut* is defined as a set of edges that disconnects the

source and sink upon removal. The *capacity* of a cut is the total capacity of all edges in the cut. A *minimum cut* of a flow network is a cut with the minimum capacity over all cuts of the flow network. Given the above mapping procedure, it is easily seen that the problem of selecting the global cut of the CEG with the minimum MEI sum now reduces to the problem of finding a minimum cut in the dual graph. By the max-flow min-cut theorem (Cormen et al., 2009), the capacity of a minimum cut in a flow network equals the maximum flow of the network. Thus, a minimum cut of the dual graph can be found by solving the maximum flow problem on the graph. Once each edge in the minimum cut is obtained, the corresponding node in the CEG from the mapping relation can be obtained and the global cut in the CEG with the minimum MEI sum can be obtained. By collecting the min-error LAC of each node in the global cut, an optimized set of LACs can be obtained.

A special case is that there may exist an edge with a negative capacity in the dual graph, which corresponds to a negative MEI of a node n in the CEG. This implies that the circuit error decreases after applying the min-error LAC of node n. Since a maximum flow algorithm cannot work on a flow network with negative capacities, a pre-processing is required, which corresponds to the pre-processing mentioned in Section 4.3. Note that the AT of node n is reduced by applying the min-error LAC of n. Such a LAC reducing both the error and the AT is highly desired. Thus, if there exists a node with a negative MEI in the CEG, the min-error LAC of the node can be directly applied to simplify the current approximate circuit and update the CEG. This process repeats until there is no negative MEI in the CEG. After that, a maximum flow algorithm can be applied.

Algorithm 9: *FindApplyOptLACSet_Flow*, a function for finding and applying an optimized LAC set to reduce delay by the maximum flow-based method.

Input: the current approximate circuit G_{apx} and an error upper bound e_b.
Output: a flag *HasSol*, denoting the existence of a valid LAC set, and a new approximate circuit G_{new}.

```
1 G_apx ⇐ PreprocessNegativeCapacity(G_apx);
2 CEG ⇐ BuildCriticalErrorGraph(G_apx);
3 DualGraph ⇐ BuildDualGraph(CEG);
4 MinCut ⇐ SolveMaxFlow(DualGraph);
5 Global cut 𝒩 ⇐ EdgeToNode(MinCut);
6 LAC set ℒ ⇐ GetMinErrorLACs(𝒩);
7 if GetError(ℒ) ≤ e_b then
      // if solutions exist
8     G_new ⇐ ApplyLACSet(G_apx, ℒ); return (true, G_new);
9 else G_new ⇐ G_apx; return (false, G_new);
```

Algorithm 9 shows the entire flow of the maximum flow-based implementation of the function *FindApplyOptLACSet*, which finds an optimized set of LACs and applies them to simplify the approximate circuit. Line 1 pre-processes the current approximate circuit G_{apx} to avoid the occurrence of negative MEIs in the CEG. It repeatedly finds the node in the CEG with the smallest MEI and applies its min-error LAC to update the

approximate circuit, until the MEIs of all nodes in the CEG of the approximate circuit are non-negative. Then, Line 2 constructs the CEG of the current approximate circuit G_{apx}. Line 3 builds the dual graph from the CEG. Line 4 solves the maximal flow problem and returns the minimum cut in the dual graph. Line 5 maps the minimum cut in the dual graph into the global cut $\mathcal{N}$ in the CEG. Line 6 obtains the set of min-error LACs $\mathcal{L}$ for all nodes in $\mathcal{N}$. Then, the error of the LAC set $\mathcal{L}$ is evaluated accurately by logic simulation. If the error of $\mathcal{L}$ does not exceed the upper bound e_b, then the LACs in $\mathcal{L}$ are applied to the approximate circuit G_{apx} pre-processed in Line 1, and Line 8 returns $HasSol = true$ and the resulting approximate circuit G_{new}. Otherwise, Line 9 returns $HasSol = false$ and the pre-processed circuit G_{apx}.

4.5.2 Priority Cut-Based Method

As mentioned in Section 4.5.1, the error estimation model used in the maximum flow-based method is not accurate. This means that the LAC set found by the maximum flow-based method can be further improved. To achieve this, a priority cut-based method is proposed in this section. For a set of nodes $\mathcal{S}$ in the CEG, $Error(\mathcal{S})$ is used to denote the error of the approximate circuit obtained by applying the min-error LACs of the nodes in set $\mathcal{S}$. It is also referred to as the *error of the node set* $\mathcal{S}$. Assume that the global cut found by the maximum flow-based method is $\mathcal{N}_{ref}$. Let $\epsilon = Error(\mathcal{N}_{ref})$. It is desired to find a better global cut $\mathcal{N}$ so that $Error(\mathcal{N}) < \epsilon$. This section first presents a basic method to do this by traversing a set of global cuts, and then a pruning method for acceleration, which leads to the priority cut-based method.

Basic Method by Traversing a Set of Global Cuts

This method traverses a set of global cuts of the CEG to find whether there exists one with a lower error than N_{ref}. Next, how to construct the set of global cuts is presented. The CEG is first modified by adding a sink node t and connecting the fanin of each PO to node t. As shown in Fig. 4.3, the fanins of the POs are n_4 and n_5, and they are connected to the sink node t. The following definition of a local cut helps the construction.

Definition 4.3. A **local cut** of a node n in the CEG is either a trivial set $\{n\}$, or a set of nodes satisfying: 1) each node in the set is a functional node, and 2) each path from a PI of the CEG to a fanin of node n passes at least one node in the set.

With the sink node t added, by Definition 4.3, a local cut of node t except $\{t\}$ corresponds to a global cut of the CEG. Thus, it is only required to obtain a set of local cuts of the sink node t. To achieve this, the method proposed in the reference (Chatterjee et al., 2006) is modified to derive a set of local cuts of each node n in the

CEG, denoted as $\Phi(n)$. The proposed method relies on an operator $\bowtie$ for merging two sets of local cuts A and B:

$$A \bowtie B = \{a \cup b | a \in A, b \in B\}.$$

Particularly, $A \bowtie B$ is empty if either A or B is empty. In addition, if there are repetitive local cuts after performing $A \bowtie B$, only one of them is kept and the others are discarded.

Example 4.4. Consider two sets of local cuts $A = \{\{n_1\}, \{n_2\}\}$ and $B = \{\{n_1\}, \{n_3\}\}$. $A \bowtie B = \{\{n_1\}, \{n_1, n_2\}, \{n_1, n_3\}, \{n_2, n_3\}\}$.

For each functional/sink n in the CEG, assume that it has r fanins $u_1, u_2, \ldots, u_r$. The set of local cuts of node n, $\Phi(n)$, is obtained recursively as follows:

$$\Phi(n) = \begin{cases} \{\{n\}\}, \text{if } n \text{ is connected to any PI;} \\ \{\{n\}\} \cup (\Phi(u_1) \bowtie \Phi(u_2) \bowtie \cdots \bowtie \Phi(u_r)), \text{otherwise.} \end{cases} \tag{4.4}$$

A brief explanation of Eq. (4.4) is as follows. For a node n connected to any PI in a CEG, assume that one of the connected PIs is x. Then, the path from x to a fanin of n has only one node, which is x, a non-functional node. Thus, there does not exist a set of nodes satisfying both Conditions (a) and (b) in Definition 4.3, and hence, only the trivial set $\{n\}$ can be a local cut of n. For a node n not connected to any PIs in the CEG, all its fanins are functional nodes. In this case, a local cut of n can be either the trivial set $\{n\}$, or a local cut obtained by merging the local cuts of the fanins of n.

Eq. (4.4) is applied to each functional/sink node in the CEG in a topological order to finally get $\Phi(t)$. After that, a set of global cuts is obtained as $\Phi(t) \backslash \{t\}$. An example of how to obtain a set of global cuts is as follows.

Example 4.5. In Fig. 4.3, a set of local cuts of each functional/sink node can be obtained by Eq. (4.4) in a topological order as follows:

$$\begin{aligned}
\Phi(n_1) =& \{\{n_1\}\}, \\
\Phi(n_2) =& \{\{n_2\}\} \cup \Phi(n_1) = \{\{n_1\}, \{n_2\}\}, \\
\Phi(n_3) =& \{\{n_3\}\} \cup \Phi(n_1) = \{\{n_1\}, \{n_3\}\}, \\
\Phi(n_4) =& \{\{n_4\}\} \cup \Phi(n_2) = \{\{n_1\}, \{n_2\}, \{n_4\}\}, \\
\Phi(n_5) =& \{\{n_5\}\} \cup (\Phi(n_2) \bowtie \Phi(n_3)) \\
=& \{\{n_1\}, \{n_1, n_2\}, \{n_1, n_3\}, \{n_2, n_3\}, \{n_5\}\},
\end{aligned}$$

$$\begin{aligned}\Phi(t) =& \{\{t\}\} \cup (\Phi(n_4) \bowtie \Phi(n_5)) = \\ & \{\{t\}, \{n_1\}, \{n_1, n_2\}, \{n_1, n_3\}, \{n_1, n_4\}, \{n_1, n_5\}, \\ & \{n_2, n_3\}, \{n_2, n_5\}, \{n_4, n_5\}, \{n_1, n_2, n_3\}, \\ & \{n_1, n_2, n_4\}, \{n_1, n_3, n_4\}, \{n_2, n_3, n_4\}\}.\end{aligned}$$

Then, a set of global cuts of the CEG is obtained as $\Phi(t) \backslash \{t\}$.

For each obtained global cut, we need to run logic simulation to obtain its error and then pick the cut with the smallest error. Unfortunately, the basic method may consider a large number of global cuts. Assume that the set of functional nodes in the CEG is $\mathcal{V}$. By Definition 4.2, a global cut is a subset of $\mathcal{V}$. In the worst case, the number of global cuts considered by the basic method, N_b, can approach the total number of subsets of $\mathcal{V}$. Thus, we have $N_b = \mathcal{O}(2^{|\mathcal{V}|})$, which makes the basic method impractical for a large circuit.

Accelerated Method

To make the basic method practical, an accelerated method based on pruning is proposed.

Consider that in a CEG where all MEIs are non-negative, two local cuts a and b are merged, and the resulting local cut is $c = a \cup b$. Assume that *Error*(c) is larger than *Error*(a) and *Error*(b). This assumption is reasonable, since c is a superset of both a and b. Thus, compared to approximating a or b with the min-error LACs, approximating c with the min-error LACs modifies more nodes in the circuit. In addition, since all MEIs are non-negative, each min-error LAC applied will increase the error. Hence, approximating c is more likely to introduce a larger error. Under this assumption, when updating the set of local cuts of node n by Eq. (4.4), only the local cuts with errors no more than ϵ are kept, *i.e.*, the minimum error obtained by the maximum flow-based method. The reason is that under the assumption, a global cut with error no more than ϵ cannot be the merge of some local cuts that include one cut with error greater than ϵ.

In practice, a new operator $\bar{\bowtie}_\epsilon$ is introduced to merge two sets of local cuts A and B with an error limit ϵ:

$$A \bar{\bowtie}_\epsilon B = \{a \cup b | a \in A, b \in B, \mathit{Error}(a \cup b) \leq \epsilon\}.$$

For a node set $\mathcal{S}$, logic simulation is used to obtain *Error*($\mathcal{S}$) accurately. In this way, only those global cuts with errors no more than ϵ are explored, and hence, a better global cut with a smaller error may be found, compared to the maximum flow-based method.

A local cut with its error no more than ϵ is called a *priority cut*. With the new operator $\bar{\bowtie}_\epsilon$, for each

functional/sink node n in the CEG, we can rewrite Eq. (4.4) to the following one to obtain a set of priority cuts of n, denoted as $\Psi(n)$:

$$\Psi(n) = \begin{cases} \{\{n\}\}, \text{if } n \text{ is connected to any PI;} \\ \{\{n\}\} \cup (\Psi(u_1) \boxtimes_\epsilon \Psi(u_2) \boxtimes_\epsilon \cdots \boxtimes_\epsilon \Psi(u_r)), \text{otherwise.} \end{cases} \tag{4.5}$$

Eq. (4.5) is also applied to each functional/sink node in the CEG in a topological order. After that, a set of global cuts with errors no more than ϵ is obtained as $\Psi(t) \backslash \{t\}$, where t is the sink node of the CEG.

In addition, for efficiency concerns, it is undesired to have too many priority cuts in $\Psi(n)$ for each n. Thus, if there are more than λ priority cuts in $\Psi(n)$, where λ is a user-specified limit, the priority cuts in $\Psi(n)$ are sorted in the ascending order of their errors and only keep the top λ priority cuts. By considering more priority cuts with a larger λ, it is likely to find a better global cut with a smaller error, which leads to a better approximate design. Meanwhile, a larger λ also causes a longer runtime. Thus, there is a quality-runtime tradeoff by applying different λ's. To decide a good choice of λ, a practical method is to select several representative benchmarks, test their quality-runtime tradeoff with various λ's, and then choose a λ leading to circuits with good quality in a short time.

Now, let us analyze the number of cuts whose errors should be evaluated by logic simulation. Assume that the maximum fanin count for all functional/sink nodes in the CEG is τ. For each functional/sink node n in the CEG, Eq. (4.5) merges the priority cuts of n's fanins. Since there are at most λ priority cuts for each fanin of n, after merging these priority cuts, $\mathcal{O}(\lambda^\tau)$ new local cuts are generated on n. It is required to evaluate the errors of these new local cuts and keep the best λ ones. Usually, τ is a small constant. For example, $\tau \leq 4$ for all circuits in the BACS (Scarabottolo et al., 2020) and ISCAS (Hansen et al., 1999) benchmark suites. Thus, for each node, the accelerated method needs to run logic simulation for $\mathcal{O}(\lambda^\tau) = \mathcal{O}(1)$ cuts. Given $|\mathcal{V}|$ nodes in the CEG, the total number of cuts that need to be simulated by the accelerated method is $\mathcal{O}(|\mathcal{V}|)$. It is much smaller than $\mathcal{O}(2^{|\mathcal{V}|})$, the number of cuts that need to be simulated by the basic method.

Entire Flow of Priority Cut-Based Method

Algorithm 10 shows the priority cut-based implementation of the function *FindApplyOptLACSet*, which finds an optimized set of LACs and applies them to simplify the approximate circuit. It also starts with a pre-processing of the input approximate circuit to avoid the occurrence of negative MEIs in the CEG (Line 1). After that, Line 2 obtains a reference solution by the maximum flow-based method to determine an important parameter, error limit ϵ, for efficient cut enumeration. If the reference solution exists, namely, the maximum flow-based

Algorithm 10: *FindApplyOptLACSet_Cut*, a function for finding and applying an optimized LAC set to reduce delay by the priority cut-based method.

Input: the current approximate circuit G_{apx}, an error upper bound e_b, and a given limit λ.
Output: a flag *HasSol*, denoting the existence of a valid LAC set, and a new approximate circuit G_{new}.

```
1  G_apx ⇐ PreprocessNegativeCapacity(G_apx);
   // get reference solution by maximum flow
2  (HasSol_ref, G_ref) ⇐ FindApplyOptLACSet_Flow(G_apx, e_b);
3  if HasSol_ref = true then error limit ε ⇐ GetError(G_ref);
4  else error limit ε ⇐ e_b;
   // improve solution by priority cut
5  CEG ⇐ BuildCriticalErrorGraph(G_apx);
6  Add a sink node t into CEG;
7  foreach functional/sink node n ∈ CEG in topo. order do
8  |  Get Ψ(n) by Eq. (4.5) with parameters ε and λ;
9  Global cut 𝒩 ⇐ the cut in Φ(t)\{t} with the smallest error;
10 LAC set ℒ ⇐ GetMinErrorLACs(𝒩);
11 if GetError(ℒ) ≤ e_b then
12 |  G_new ⇐ ApplyLACSet(G_apx, ℒ); return (true, G_new);
13 Local cut 𝒩' ⇐ the local cut with the smallest error from all Ψ(n)'s, where n is a functional node in
     CEG;
14 LAC set ℒ' ⇐ GetMinErrorLACs(𝒩');
15 if GetError(ℒ') ≤ e_b then
16 |  G_new ⇐ ApplyLACSet(G_apx, ℒ'); return (true, G_new);
17 G_new ⇐ G_apx; return (false, G_new);
```

method can find a set of LACs to reduce the circuit delay without violating the error bound e_b, then Line 3 sets ϵ as the error of the approximate circuit G_{ref} produced by the maximum flow-based method. Otherwise, Line 4 sets ϵ as e_b. Then, it is tried to improve the reference solution by efficiently enumerating priority cuts. To do this, Line 5 builds a CEG from the pre-processed approximate circuit G_{apx}, and Line 6 adds a sink node t to the CEG. Then, Line 7 traverses each functional/sink node in the CEG following a topological order, and a set of priority cuts of each node n, $\Psi(n)$, is obtained by Eq. (4.5) based on the error limit ϵ and the parameter λ (Line 8). Next, Lines 9–10 derive the global cut with the smallest error from $\Psi(t)\backslash\{t\}$ and the set of min-error LACs $\mathcal{L}$ for all the nodes in the global cut. If the error of the LAC set $\mathcal{L}$ does not exceed the error bound e_b (Line 11), then Line 12 applies the LACs in $\mathcal{L}$ to further approximate the circuit G_{apx}, and returns $HasSol = true$ and the resulting circuit G_{new}. Otherwise, when there is no global cut with error no more than e_b, the local cuts of all functional nodes in the CEG are further explored, since the LACs on local cuts may contribute to delay reduction too. Line 13 selects the best local cut $\mathcal{N}'$ with the smallest error from all $\Psi(n)$'s, where n is a functional node in the CEG. Line 14 obtains the set of min-error LACs $\mathcal{L}'$ for all nodes in $\mathcal{N}'$. If the error of the LAC set $\mathcal{L}'$ does not exceed the error bound e_b, then $\mathcal{L}'$ is used to simplify G_{apx}, and Line 16 returns $HasSol = true$ and the resulting circuit G_{new}. However, it is also possible that there is no global or local cut with error no more than e_b. In this case, Line 17 returns $HasSol = false$ and the pre-processed circuit G_{apx}.

Compared to the maximum flow-based method, the priority cut-based method has some computation overhead. By analyzing Algorithm 10, the overhead mainly lies in the error evaluation of many local cuts by logic simulation, which helps build the set of priority cuts for each node (Lines 7–8 of Algorithm 10). As discussed at the end of Section 4.5.2, the number of these local cuts is $\mathcal{O}(|\mathcal{V}|)$. Assume that the number of input patterns used in each logic simulation is M and the number of nodes in the entire circuit is W. Then, the time complexity to simulate the circuit to get the error of one cut is $\mathcal{O}(MW)$. As the simulation is performed for $\mathcal{O}(|\mathcal{V}|)$ cuts to get their errors, the computation overhead of the priority cut-based method is $\mathcal{O}(MW|\mathcal{V}|)$.

4.6 Experimental Results

This section presents the experimental results.

4.6.1 Experimental Setup

Table 4.1: Benchmark circuit information.

Benchmark suite	Circuit	#I/Os	AIG		Gate-netlist	
			Size	Depth	Area	Delay
ISCAS85	c880	60/26	313	22	231.7	0.38
	c1355	41/32	390	16	429.1	0.45
	c1908	33/25	367	25	378.8	0.61
	c2670	233/140	579	17	534.1	0.40
	c3540	50/22	937	32	716.3	0.78
	c5315	178/123	1306	28	951.0	0.57
	c7552	207/108	1469	26	1030.2	1.06
BACS	absdiff	16/9	104	14	101.9	0.30
	add32	64/33	302	20	254.0	0.43
	buttfly	32/34	227	31	212.0	0.50
	mac	12/8	124	20	142.8	0.37
	mult8	16/16	470	44	496.4	0.88
	mult16	32/32	2033	41	2337.9	0.84
EPFL	add128	256/129	1019	314	982.9	4.83
	barshift	135/128	2688	14	1945.0	0.85
	divisor	128/128	23667	4473	19949.5	89.78
	log2	32/32	38540	419	26422.8	11.56
	max	512/130	2686	549	2456.2	10.98
	mult64	128/128	33242	326	22401.5	6.87
	sine	24/25	7044	180	5334.4	4.50
	sqrt	128/64	21951	4591	19035.5	128.16
	square	64/128	20030	296	14394.6	5.88

All the experiments are carried out on a computer with an Intel Xeon Gold 6146 CPU (3.20GHz) and 256

GB memory running Ubuntu 20.04. HEDALS is implemented in C++ and tested with a single thread of the CPU. The average errors are measured through logic simulation. Assume that the input patterns are uniformly distributed, although other input distributions can also be handled. For each logic simulation, 100,000 input vectors are randomly generated, which are sufficient to obtain average errors with a high accuracy (Ma et al., 2021; Su et al., 2022). The delay-oriented synthesis and mapping in HEDALS (*i.e.*, the function *Synthesize-AndMap* in Algorithm 7) are performed by ABC (Mishchenko et al., 2022). The ABC script "*resyn*; *resyn2*" is applied 6 times for technology-independent synthesis and the ABC command *map* for technology mapping using the Nangate 45nm open cell library (Nangate, Inc., 2022). ABC is also used to report the circuit area in μm^2 and delay in ns after mapping, where the delay is reported by the STA command *stime*. Delay and area ratios are used to evaluate the cost of the approximate designs. The delay ratio (*resp.* area ratio) is defined as the ratio of the delay (*resp.* area) of the approximate circuit over that of the original one. Obviously, smaller delay and area ratios are preferred.

Table 4.1 lists benchmarks used in experiments, including several circuits from the ISCAS85 benchmark suite (Hansen et al., 1999), all arithmetic circuits from the BACS benchmark suite (Scarabottolo et al., 2020), and some largest circuits from the EPFL benchmark suite (EPFL LSI Lab, 2021). Both the AIG and gate-netlist representations are considered, and they are optimized by ABC to fully minimize their delays. Specifically, to produce the AIG representation of a benchmark, the benchmark is first converted into AIG with the ABC command *strash*, and then the AIG is fully optimized by repeatedly applying the delay-oriented synthesis script *resyn2* until its quality cannot be further improved. The sizes and depths of the optimized AIGs are listed in columns 4 and 5 of Table 4.1. To produce the gate-netlist representation of a benchmark, it is first converted into an AIG with *strash*, and the AIG delay is fully optimized in the same way as the processing for the AIG representation. Then, the delay-oriented technology mapping command *map* is applied to the AIG. The last two columns of Table 4.1 list the areas and delays of the gate netlists.

In what follows, several sets of experiments are designed to study the performance of HEDALS. They involve various types of LACs, circuit representations, and error constraints.

4.6.2 Study under the NMED Constraint

This section performs experiments under the NMED constraint on the BACS benchmarks listed in Table 4.1. NMED is a commonly-used error metric for arithmetic circuits. HEDALS is applied on the AIG representation of a circuit, but finally, the AIG is mapped into a gate netlist for area and delay evaluation.

Accuracy and Impact of the Linear Error Estimation Models

Section 4.5.1 proposes a linear model, Eq. (4.3), to estimate the error of a LAC set. Its accuracy is first evaluated on the adder and multipliers in the BACS benchmark suite, *i.e.*, *adder32*, *mult8*, and *mult16*. The maximum flow-based HEDALS is run using the linear error estimation model. The applied LAC is the constant LAC (Shin and Gupta, 2011), *i.e.*, replacing a node by a constant 0 or 1. For each benchmark, the first 7 iterations in the HEDALS flow are considered. For each optimized LAC set found in each iteration, its *estimated error* (*EER*) obtained by Eq. (4.3) is compared with the *actual error* (*AER*) obtained by logic simulation. The same set of 100,000 input vectors is used to calculate the EER and AER. As shown in Fig. 4.5, except iterations 4 and 6 for *mult8* and iteration 7 for *mult16*, the EER is close to AER for all the 7 iterations of the three circuits.

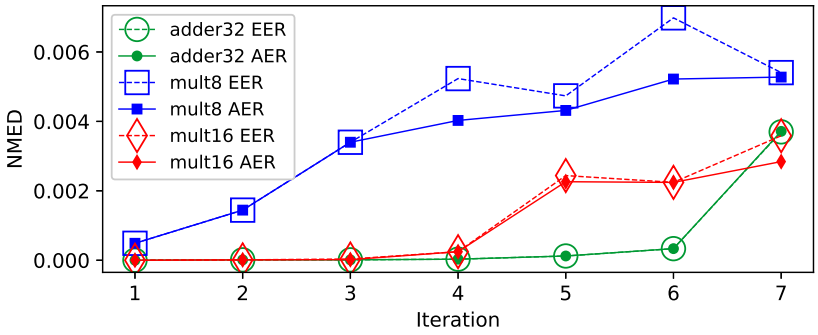


Figure 4.5: Comparison of estimated error (EER) and actual error (AER). The error metric is NMED.

In the prior work (Zhou et al., 2018), a less accurate linear error estimation model shown in Eq. (4.2) is proposed. The impacts of this model are further compared with the proposed more accurate model based on Eq. (4.3). Each point in each AER curve in Fig. 4.5 is considered, which corresponds to an intermediate approximate design. Eqs. (4.2) and (4.3) are applied, respectively, to each intermediate design and obtain their resulting optimized LAC sets. Table 4.2 lists all the circuits and their iterations where the LAC set found using Eq. (4.2) differs from that found using (4.3). It also lists the AERs of those different LAC sets. It can be seen see that the proposed model, Eq. (4.3), can lead to better LAC sets with smaller AERs. This section also compares the final circuit quality achieved by both models under two NMED bounds, 0.3% and 3.0%. Compared to using Eq. (4.2), using Eq. (4.3) reduces more delay by 7.9%, 1.7%, and 0.8%, on average, for *adder32*, *mult8*, and *mult16*, respectively, while the average circuit area does not increase.

Table 4.2: All different optimized LAC sets found by different linear error models in the first 7 iterations of HEDALS. The **bold** entries mean that Eq. (4.3) leads to better LAC sets with smaller AERs than Eq. (4.2).

Circuit	Iteration	AER of optimized LAC set by Eq. (4.2)	AER of optimized LAC set by Eq. (4.3)
adder32	3	1.45E-05	**1.14E-05**
adder32	6	3.75E-04	**3.34E-04**
mult8	6	8.10E-03	**5.22E-03**
mult16	6	2.25E-03	**2.24E-03**

Comparing Maximum Flow-Based Method with Priority Cut-Based Method

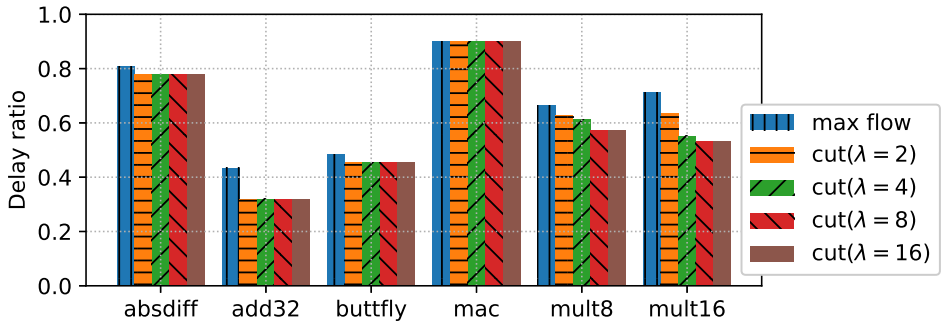
In the HEDALS framework, a maximum flow-based method and a priority cut-based method are designed to solve Problem 4.2. Their performance is compared in this section with an NMED bound of 0.005. The applied LAC is the constant LAC.

Fig. 4.6 compares the delay ratio, area ratio, and runtime of the maximum flow-based and priority cut-based methods. Different limits λ's are used in the priority cut-based method. For each circuit, the runtime is normalized to that of the priority cut-based method with $\lambda = 16$.

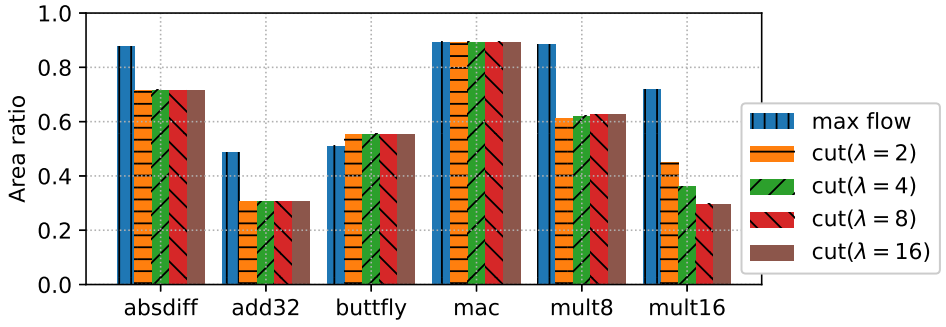
As shown in Figs. 4.6(a) and 4.6(b), the priority cut-based method reduces more delay and area than the maximum flow-based method on most benchmarks. The reason is that using priority cuts, a set of LACs causing a smaller error can be found in each iteration of the HEDALS flow. Then, the error increase of each iteration is smaller, leading to more iterations and hence, more sets of LACs applied to simplify the circuit. However, there are two exceptions. The first is *mac*. Its approximate designs generated by both methods have the same delay and area. Actually, in each iteration during the synthesis of *mac*, the set of LACs found by both methods are exactly the same, and hence, the final approximate circuits are identical. The second one is *buttfly*, for which the priority cut-based method produces approximate circuits with smaller delays but larger areas. In terms of runtime, the priority cut-based method takes a longer runtime, since it calls the maximum flow-based method to generate a reference solution.

In terms of the influence of the parameter λ, Fig. 4.6(a) shows that delay ratio decreases or stays the same as λ increases. As shown in Fig. 4.6(b), area ratio behaves similarly as delay ratio with different λ's. It is reasonable since with a larger λ, more priority cuts in the CEG are considered. Thus, it is more likely to find a better global cut with a smaller error. This leads to more approximation iterations and hence, more delay and area reduction. Furthermore, as λ changes from 8 to 16, the delay ratios of all circuits remain unchanged. It implies that by keeping at most 8 priority cuts for each node in the CEG, good enough approximate circuits with small delays can be found. In terms of efficiency, Fig. 4.6(c) shows that the runtime increases with λ. This is reasonable since with a larger λ, more priority cuts are considered for each node, which takes a longer

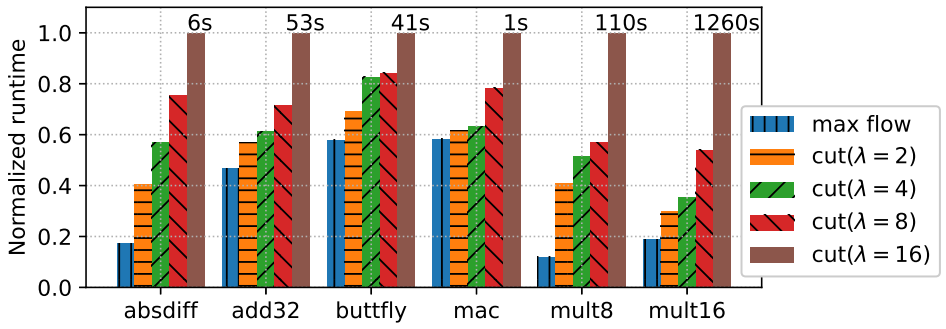
runtime. Considering delay, area, and runtime, the priority cut-based method with $\lambda = 8$ can achieve good delay and area savings in a short time. Thus, in the remaining experiments, this implementation is chosen for HEDALS unless otherwise specified.



(a) Delay comparison for various benchmarks and methods.



(b) Area comparison for various benchmarks and methods.

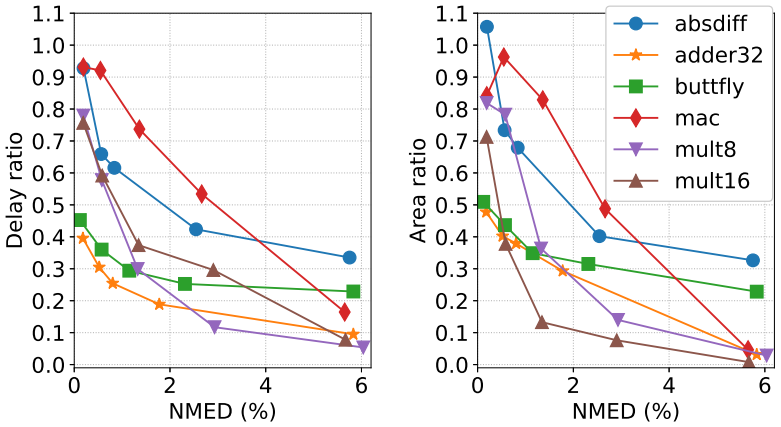


(c) Runtime comparison for various benchmarks and methods.

Figure 4.6: Comparison between the maximum flow-based and priority cut-based methods on the BACS benchmark suite under an NMED bound of 0.005.

Quality-NMED Tradeoff of Approximate Designs

This section studies the quality-NMED tradeoff of the approximate circuits synthesized by HEDALS. To show



(a) Delay-NMED tradeoff. **(b)** Area-NMED tradeoff.

Figure 4.7: Quality-NMED tradeoff on the BACS benchmark suite.

that HEDALS can support various LACs, another LAC is selected, the ALSRAC LAC (Meng et al., 2020).

HEDALS is run on each BACS benchmark under 5 NMED bounds, *i.e.*, $\frac{1}{2^9-1} \approx 0.20\%$, $\frac{3}{2^9-1} \approx 0.59\%$, $\frac{7}{2^9-1} \approx 1.37\%$, $\frac{15}{2^9-1} \approx 2.94\%$, and $\frac{31}{2^9-1} \approx 6.07\%$. Fig. 4.7(a) plots the delay ratio-NMED curves. For each circuit, the delay decreases monotonically with the NMED. When the NMED reaches 6.07% (see the rightmost points on the curves), the delay ratio of different circuits ranges from 5% (*i.e.*, *mult8*) to 34% (*i.e.*, *absdiff*), that is, the circuit delay is significantly reduced by 66% $\sim$ 95%. Although area saving is a byproduct of HEDALS, the area ratio-NMED curves are also plotted in Fig. 4.7(b). From Fig. 4.7(b), areas of most approximate circuits are less than those of the exact counterparts. Moreover, the area decreases with the NMED for all the circuits except *mac*. For *mac*, when the NMED changes from 0.20% to 0.59%, its area increases. It is because HEDALS works on the AIG representation of a circuit in this experiment, but the area and delay are evaluated after technology mapping. As the NMED changes from 0.20% to 0.59% for *mac*, the AIG size keeps unchanged and the depth decreases, which is expected. However, its area increases after mapping due to an occasional inconsistency between the quality change trend of an AIG and that of a mapped gate netlist. When the NMED approaches 6.07% (see the rightmost points on the curves), the area ratio of different circuits varies from 1% (*i.e.*, *mult16*) to 33% (*i.e.*, *absdiff*), that is, the circuit area is dramatically reduced by 67% $\sim$ 99%.

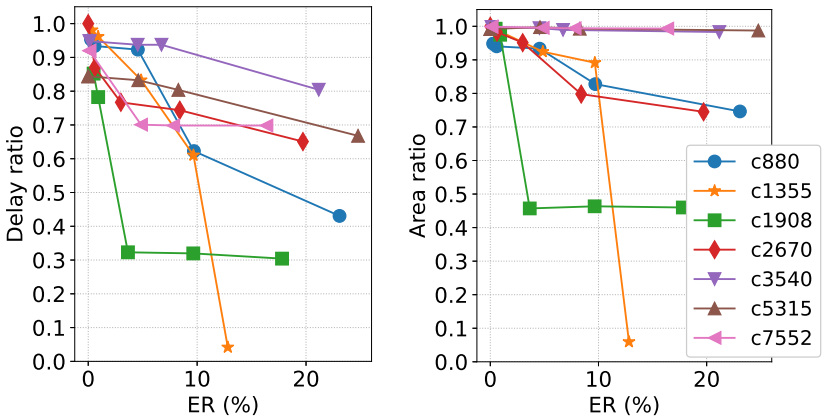
HEDALS is also compared with a state-of-the-art ALS flow, *BLASYS* (Ma et al., 2021), which is an open-source area-oriented flow. BLASYS is run with 48 threads of the CPU, and its runtime is reported as the total runtime of all threads. For a more fair comparison with the proposed delay-oriented HEDALS flow, BLASYS is slightly modified to enhance its ability in reducing delay. Specifically, the modified BLASYS applies the same delay-oriented synthesis and mapping process used in HEDALS, *i.e.*, applying the ABC script "resyn;

Table 4.3: Comparison between HEDALS and BLASYS under the NMED constraint. The **bold** values mean that HEDALS outperforms BLASYS.

Circuit	NMED bound	Delay ratio		Area ratio		Runtime/min	
		HEDALS	BLASYS	HEDALS	BLASYS	HEDALS	BLASYS
absdiff	0.59%	**65.9%**	103.9%	**73.4%**	83.5%	**0.09**	2.84
	2.94%	**42.3%**	98.3%	**40.2%**	74.7%	**0.14**	6.09
adder32	0.59%	**30.5%**	43.6%	40.2%	31.9%	**0.54**	39.00
	2.94%	**18.9%**	24.3%	**29.3%**	33.1%	**0.55**	41.14
buttfly	0.59%	**36.0%**	45.7%	**43.7%**	61.7%	**0.29**	27.08
	2.94%	**25.3%**	37.5%	**31.5%**	41.2%	**0.41**	29.63
mac	0.59%	**92.1%**	97.6%	96.3%	95.2%	**0.05**	0.22
	2.94%	**53.4%**	104.8%	**48.8%**	72.4%	**0.14**	2.61
mult8	0.59%	**57.8%**	60.7%	78.2%	34.5%	**1.18**	1168.26
	2.94%	**11.7%**	34.0%	**14.0%**	16.2%	**2.10**	1235.36
mult16	0.59%	**59.2%**	70.6%	37.8%	15.8%	**21.59**	2835.34
	2.94%	**29.6%**	52.1%	**7.6%**	7.8%	**23.06**	3007.71
Average		**43.6%**	64.4%	**45.1%**	47.3%	**4.18**	699.61

resyn2" 6 times, followed by the ABC command *map*. The approximate designs generated by HEDALS and BLASYS are compared under two NMED bounds, 0.59% and 2.94%. The results are listed in Table 4.3. For all benchmarks, the delay ratio of HEDALS is much smaller than that of BLASYS. On average, HEDALS further reduces the delay ratio by a relative value of 32.3% over BLASYS. It is not surprising that BLASYS is not good at reducing delay. As an area-oriented ALS method, BLASYS does not specifically optimize the nodes on the critical paths. In contrast, HEDALS reduces more delay due to a direct effort on the nodes on the critical paths. Thus, HEDALS can provide a better solution when delay is the primary optimization goal. Furthermore, since area reduction is just a side effect of HEDALS, it may not be as much as that of BLASYS. However, it is worth noting that for all benchmarks under some NMED bounds, HEDALS can even reduce more area. The reason is that BLASYS partitions a circuit into sub-circuits and reduces the area of each sub-circuit separately. This method may miss some area reduction opportunities, such as the simplification across two sub-circuits. On the contrary, HEDALS does not partition the circuit and can consider some opportunities that BLASYS misses, hence reducing more area than BLASYS sometimes. As for efficiency, HEDALS is much faster than BLASYS on all benchmarks. On average, HEDALS accelerates by 167× over BLASYS. The acceleration arises from two aspects. First, although both HEDALS and BLASYS simplify circuits iteratively, HEDALS has fewer iterations. Its reason is that HEDALS applies multiple LACs in each iteration, while BLASYS only applies one. Hence, HEDALS modifies more sub-circuits in each iteration and approaches the error bound faster. Second, the runtime of HEDALS for each iteration is much shorter, since it only focuses on the nodes on the critical graph of the circuit and only considers the min-error LACs of these nodes. Besides, it prunes LAC sets with large errors by the priority cut-based method.

4.6.3 Study under ER Constraint



(a) Delay-ER tradeoff. **(b)** Area-ER tradeoff.

Figure 4.8: Quality-ER tradeoff on the ISCAS benchmark suite.

This section studies the performance of HEDALS on the 7 ISCAS85 circuits listed in Table 4.1 under the ER constraint. ER measures the erroneous probability of a circuit. It is suitable for evaluating the accuracy of circuits such as classifiers, controllers, and error correctors (*e.g.*, *c1355* and *c1908*). It is also widely used as an additional error metric for arithmetic circuits (*e.g.*, *c7552*) (Liu et al., 2014a) and arithmetic logic units (*e.g.*, *c880*, *c2670*, *c3540*, and *c5315*) (Venkataramani et al., 2013). To further show the wide applicability, HEDALS is applied to the gate-netlist representation of these benchmarks with the constant LAC.

HEDALS is run on each benchmark under 5 ER bounds, 0.5%, 1%, 5%, 10%, and 25%. Fig. 4.8(a) plots the delay ratio-ER curves of the approximate circuits generated by HEDALS. For all circuits, the delay decreases monotonically with the ER. When the ER reaches 25% (see the rightmost points on the curves), the delay ratio of different circuits ranges from 4% (*i.e.*, *c1355*) to 80% (*i.e.*, *c3540*), that is, the circuit delay is significantly reduced by 20% $\sim$ 96%. The area ratio-ER curves are also plotted in Fig. 4.8(b). For all approximate circuits, their areas are smaller than the areas of the exact counterparts. Moreover, the area decreases monotonically with the ER for all circuits. For some circuits such as *c3540*, *c5315*, and *c7552*, the area is almost unchanged. However, for some other circuits such as *c1355* and *c1908*, the area also reduces by a large amount. In particular, HEDALS can reduce the area by 94% and the delay by 96% simultaneously on the benchmark *c1355* under an ER of 12.8%.

HEDALS is also compared with BLASYS, where the setup of BLASYS is the same as that in Section 4.6.2. The approximate designs generated by HEDALS and BLASYS are compared under two ER bounds, 0.5% and 5%. The results are listed in Table 4.4. The delay ratio of HEDALS is always smaller than that of BLASYS

except for the case of *c2670* under an NMED bound of 0.005%. In this case, HEDALS cannot simplify *c2670*, while BLASYS can. On average, HEDALS further reduces the delay ratio by a relative value of 6.1% over BLASYS. Table 4.4 also shows that HEDALS reduces less area than BLASYS, but it is guaranteed that the area of an approximate circuit produced by HEDALS is no larger than that of the exact counterpart. As for efficiency, HEDALS is much faster than BLASYS on all circuits. On average, HEDALS accelerates by 9799× over BLASYS.

Table 4.4: Comparison between HEDALS and BLASYS under the ER constraint. The **bold** values mean that HEDALS is better than BLASYS.

Circuit	ER bound	Delay ratio		Area ratio		Runtime/min	
		HEDALS	BLASYS	HEDALS	BLASYS	HEDALS	BLASYS
c880	0.5%	**95.3%**	98.6%	94.8%	89.9%	**0.01**	24.1
	5.0%	**92.3%**	93.9%	93.3%	75.0%	**0.04**	48.6
c1355	0.5%	**98.1%**	120.8%	99.4%	98.3%	**0.03**	70.6
	5.0%	**83.3%**	94.1%	**92.4%**	95.5%	**0.05**	200.3
c1908	0.5%	**85.2%**	90.6%	99.3%	90.9%	**0.02**	55.5
	5.0%	**32.3%**	33.7%	45.7%	41.9%	**0.16**	385.4
c2670	0.5%	100.0%	79.6%	100.0%	64.5%	**0.03**	297.6
	5.0%	**76.7%**	81.5%	95.1%	63.7%	**0.17**	347.7
c3540	0.5%	**94.8%**	99.7%	99.8%	92.0%	**0.02**	445.4
	5.0%	**93.7%**	101.6%	99.4%	83.0%	**0.04**	1366.0
c5315	0.5%	**84.4%**	88.2%	99.2%	96.5%	**0.05**	2267.6
	5.0%	**83.2%**	86.9%	99.7%	97.5%	**0.15**	2538.6
c7552	0.5%	**92.0%**	93.7%	99.9%	84.4%	**0.07**	1362.9
	5.0%	**70.1%**	95.4%	99.5%	81.7%	**0.30**	1564.5
Average		**84.4%**	89.9%	94.1%	82.5%	**0.08**	783.9

4.6.4 Study under MHD Constraint

To show its scalability, HEDALS is run on the large EPFL benchmarks listed in Table 4.1 under the NMHD constraint. HEDALS is applied on the AIG representation of a circuit, and the applied LAC is ALSRAC. Since these benchmarks are large, the priority cut-based method with $\lambda = 1$ is applied to save runtime. HEDALS is also compared with BLASYS, but the BLASYS program is not re-run due to its long runtime. For example, as reported in the reference (Ma et al., 2021), it takes BLASYS about 19 days to generate an approximate circuit for the benchmark *sine* with 7044 AIG nodes under an NMHD bound of 5%. Instead, data from the reference (Ma et al., 2021) is directly used for comparison. Although a different 65nm standard cell library is used in the reference (Ma et al., 2021), the area and delay ratios from the reference (Ma et al., 2021) are still good references.

The result of HEDALS and BLASYS are compared on the EPFL benchmarks under the same two NMHD

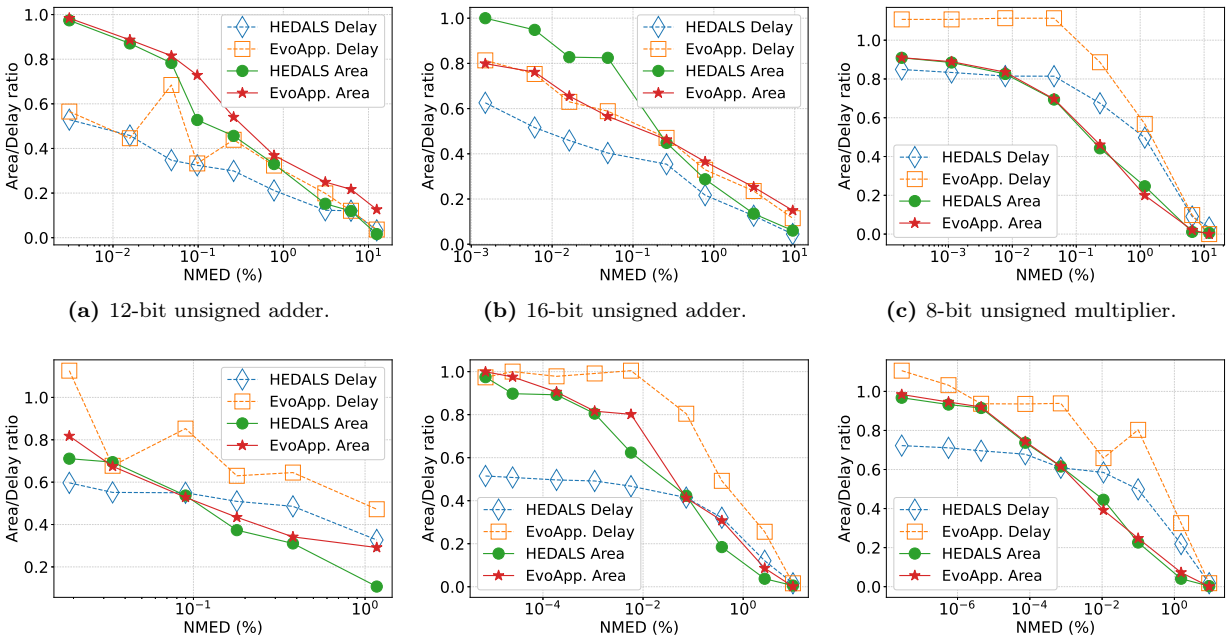
bounds used in the reference (Ma et al., 2021), 5% and 10%. As shown in Table 4.5, the circuit delay is significantly reduced by HEDALS. On average, the delay ratio of HEDALS is 45.3%, which is about half of that of BLASYS. Particularly, HEDALS saves much delay on *divisor*. It reduces 97.7% and 99.2% delay under NMHD bounds of 5% and 10%, respectively. Meanwhile, the average area ratio of HEDALS is 69.2%, which is relatively 19.2% smaller than BLASYS. Furthermore, HEDALS is very efficient: its runtime for most designs is less than one hour.

Table 4.5: Performance of HEDALS on the large EPFL circuits. The **bold** values mean that HEDALS outperforms BLASYS. Data of BLASYS are from the reference (Ma et al., 2021). N/A means that the corresponding data is not reported in the reference (Ma et al., 2021).

Circuit	NMHD bound	Delay ratio		Area ratio		Runtime/min	
		HEDALS	BLASYS	HEDALS	BLASYS	HEDALS	BLASYS
add128	5%	**35.1%**	90.8%	**81.3%**	89.4%	**0.3**	340.3
	10%	**15.6%**	80.9%	**72.6%**	79.4%	0.4	N/A
barshift	5%	**91.8%**	105.6%	**87.1%**	95.8%	**19.2**	3510
	10%	**83.4%**	88.5%	**84.0%**	90.0%	34.6	N/A
divisor	5%	**2.3%**	91.6%	**6.3%**	85.9%	62.8	N/A
	10%	**0.8%**	73.0%	**4.3%**	76.2%	89.3	N/A
log2	5%	**77.8%**	100.5%	96.8%	92.9%	19.1	N/A
	10%	**70.5%**	78.2%	96.6%	82.1%	53.0	N/A
max	5%	**21.1%**	114.3%	**34.5%**	91.0%	6.2	N/A
	10%	**20.5%**	94.3%	**31.1%**	77.6%	4.5	N/A
mult64	5%	**59.0%**	99.4%	88.8%	87.7%	14.7	N/A
	10%	**53.3%**	93.8%	90.2%	80.5%	21.2	N/A
sine	5%	**76.5%**	93.1%	94.9%	84.3%	**1.3**	27849.3
	10%	**70.6%**	79.9%	93.6%	71.7%	2.1	N/A
sqrt	5%	50.6%	N/A	52.4%	N/A	332.0	N/A
	10%	40.5%	N/A	42.1%	N/A	394.3	N/A
square	5%	**25.0%**	85.8%	97.9%	95.8%	14.5	N/A
	10%	**21.6%**	75.5%	90.3%	88.5%	19.4	N/A
Average w/o sqrt		**45.3%**	90.3%	**69.2%**	85.6%	60.5	N/A

4.6.5 Study on Adders and Multipliers

This section generates various approximate adders and multipliers by HEDALS, and compares them with the circuits in *EvoApproxLib* (version 2022) (Mrazek et al., 2022), a widely-used library of approximate arithmetic circuits, which collects a series of approximate circuits from the reference (Mrazek, 2022; Mrazek et al., 2018, 2017, 2016), produced by evolutionary algorithm-based ALS methods. The selected benchmarks are some largest adders and multipliers in EvoApproxLib, *i.e.*, 12-bit and 16-bit unsigned adders, and 8-bit, 11-bit, 12-bit, and 16-bit unsigned multipliers. For each benchmark, the approximate designs are selected from EvoApproxLib under the NMED (called mean absolute error in EvoApproxLib) constraint. Since the published



(a) 12-bit unsigned adder. **(b)** 16-bit unsigned adder. **(c)** 8-bit unsigned multiplier.

(d) 11-bit unsigned multiplier. **(e)** 12-bit unsigned multiplier. **(f)** 16-bit unsigned multiplier.

Figure 4.9: Comparison between the approximate designs synthesized by HEDALS and those from the EvoApproxLib on delay and area ratios versus NMED.

designs in EvoApproxLib are synthesized with a different standard cell library, the EvoApproxLib designs are re-synthesized and mapped into the Nangate 45nm library with the same delay-oriented synthesis and mapping process used in HEDALS. Then, to compare with each design in EvoApproxLib, HEDALS is applied to generate an approximate circuit with the NMED bound as the NMED of the EvoApproxLib design. HEDALS works on the AIG representation using the ALSRAC LAC. Since sometimes there is a large gap between the error of an approximate design C generated by HEDALS and the error bound, the design C is further simplified with the ALSRAC LAC until the error bound is reached. This post-processing can further reduce circuit area without increasing delay.

The results are shown in Fig. 4.9. Each sub-figure of Fig. 4.9 plots the delay ratio-NMED and area ratio-NMED curves of the approximate designs synthesized by HEDALS and those from the EvoApproxLib for a benchmark. HEDALS always reduces more delay than EvoApproxLib, except for two cases, the 12-bit adder under an NMED of 0.018% and the 8-bit multiplier under an NMED of 25%. Moreover, the delays of some EvoApproxLib circuits sometimes exceed those of the corresponding exact circuits, while HEDALS guarantees that the delay of an approximate circuit is always smaller than that of the corresponding exact design. It is not surprising because HEDALS is designed for delay optimization while the EvoApproxLib method is not. Furthermore, HEDALS saves more area than EvoApproxLib on the 12-bit adder and achieves competitive area

savings on the rest benchmarks. It is because the EvoApproxLib designs are generated by the evolutionary algorithm, which features randomness and may miss some area reduction opportunities, while HEDALS uses a different greedy strategy to select the LAC sets and may capture some area reduction opportunities missed by the evolutionary algorithm.

4.6.6 Comparison of HEDALS Performance with Different LAC Types and Circuit Representations

Table 4.6: Comparison of HEDALS performance with different LAC types and circuit representations under ER and NMED constraints. "DR" means delay ratio, and "AR" means area ratio. The best choice among the three combinations of LAC type and circuit representation is highlighted in **bold**.

Circuit	LAC+representation	ER≤0.005		ER≤0.01		ER≤0.05		ER≤0.1		Average	
		DR	AR	DR	AR	DR	AR	DR	AR	DR	AR
adder32	CONST+AIG	**69.9%**	**96.4%**	69.9%	96.4%	**66.8%**	95.9%	65.7%	**93.6%**	68.1%	**95.6%**
	CONST+GATE	99.4%	96.5%	99.4%	96.5%	89.1%	95.5%	89.1%	95.5%	94.2%	96.0%
	ALSRAC+AIG	**69.9%**	**96.4%**	**66.2%**	**96.2%**	69.3%	**95.4%**	**61.1%**	95.2%	**66.6%**	95.8%
mult8	CONST+AIG	97.0%	99.2%	97.0%	99.2%	97.0%	99.2%	95.0%	**95.0%**	96.5%	98.2%
	CONST+GATE	100.0%	100.0%	100.0%	100.0%	99.6%	99.8%	94.6%	99.7%	98.6%	99.9%
	ALSRAC+AIG	**94.4%**	**99.0%**	**94.4%**	**99.0%**	**93.3%**	**86.7%**	**89.6%**	99.5%	**92.9%**	**96.0%**

Circuit	LAC+representation	NMED≤0.005%		NMED≤0.01%		NMED≤0.05%		NMED≤0.1%		Average	
		DR	AR	DR	AR	DR	AR	DR	AR	DR	AR
adder32	CONST+AIG	**62.7%**	**60.3%**	**59.8%**	**59.4%**	58.5%	47.2%	**47.6%**	**37.5%**	57.1%	**51.1%**
	CONST+GATE	70.8%	91.0%	60.1%	73.8%	57.1%	74.7%	57.1%	74.7%	61.2%	78.5%
	ALSRAC+AIG	**62.7%**	**60.3%**	**59.8%**	**59.4%**	**53.1%**	**45.9%**	52.5%	38.7%	**57.0%**	**51.1%**
mult8	CONST+AIG	97.0%	99.2%	97.0%	99.2%	91.8%	**93.8%**	82.7%	92.7%	92.1%	96.2%
	CONST+GATE	100.0%	100.0%	100.0%	100.0%	90.5%	95.3%	89.5%	94.3%	95.0%	97.4%
	ALSRAC+AIG	**95.5%**	**96.1%**	**96.2%**	**98.2%**	**90.0%**	96.9%	**80.5%**	**87.0%**	**90.6%**	**94.5%**

Since HEDALS can be applied with different LAC types and circuit representations, it is also interesting to study the performance of HEDALS with different combinations of LAC type and circuit representation. This study is performed on two arithmetic circuits, *adder32* and *mult8*, and two error metrics, ER and NMED. The selected ER bounds are 0.005, 0.01, 0.05, and 0.1, and the selected NMED bounds are 0.005%, 0.01%, 0.05%, and 0.1%. For each ER or NMED bound, this section considers constant and ALSRAC LACs, and AIG and gate-netlist representations. Since the ALSRAC LAC cannot work on gate netlists (Meng et al., 2020), only 3 combinations are studied, *i.e.*, *CONST+AIG*, *CONST+GATE*, and *ALSRAC+AIG*. The performance of HEDALS with the 3 combinations of LAC type and circuit representation is shown in Table 4.6. The delay and area ratios of each approximate circuit generated by HEDALS with each combination are reported. Comparing

CONST+AIG with *CONST+GATE*, the former reduces more delay and area in most cases. One possible reason is that there are more nodes in an AIG than in a gate netlist. Hence, an AIG has more candidate LACs for circuit simplification, leading to better approximate designs. Comparing *ALSRAC+AIG* with *CONST+AIG*, the former outperforms the latter with smaller delay and area ratios in most cases. It is not surprising since the ALSRAC LAC is based on signal resubstitution, which is a more fine-grained circuit simplification technique than constant replacement. To sum up, *ALSRAC+AIG* is the best choice for the two circuits.

4.7 Summary

This chapter proposes HEDALS, a highly efficient delay-driven approximate logic synthesis framework. Its basic idea is to establish a critical graph of a target circuit and find an optimized LAC set based on the graph, which is then applied to shorten all the critical paths simultaneously. A maximum flow-based method and a priority cut-based method are proposed to find an optimized LAC set. The former is faster, while the latter can lead to better approximate designs. The experimental results on a wide range of benchmarks show that HEDALS outperforms the state-of-the-art ALS approaches. Furthermore, it supports various LACs, circuit representations, and average error metrics. Thus, HEDALS is a promising solution for synthesizing approximate circuits with minimized delays.


20006301

Chapter 5

MECALS: Efficient Maximum Error Checking Technique for Approximate Logic Synthesis

This chapter presents an efficient method called MECALS to check the maximum errors of LACs in the ALS flow.

5.1 Motivations and Overview

As mentioned in Section 1.4.2, the most time-consuming step in the ALS flow is evaluating errors. There are typically two types of error metrics used in ALS (Scarabottolo et al., 2020), *i.e.*, average and maximum errors. Average errors, such as ER and MED, measure the average deviation between the outputs of exact and approximate circuits. Maximum errors, such as *worst-case error* (*WCE*) and *maximum square error* (*MaxSE*), compute the maximum deviation between the outputs of exact and approximate circuits over all input patterns. Many ALS methods are proposed for different error metrics.

Some ALS methods focus on average error constraint (Venkataramani et al., 2013; Wu and Qian, 2016; Hashemi et al., 2018; Meng et al., 2020). Venkataramani et al. (2013) proposed to substitute a signal with another signal with a similar function in the circuit under ER or MED constraint. Wu and Qian (2016) proposed to delete some literals from the Boolean expression of a node in the circuit under ER constraint. Hashemi et al. (2018) utilized Boolean matrix factorization to approximate sub-circuits under MED or MRED

constraint. Meng et al. (2020) proposed to perform approximate resubstitution based on approximate care sets under ER, MED, or MRED constraint.

Some ALS methods deal with maximum error constraint (Venkataramani et al., 2012; Scarabottolo et al., 2018; Witschen et al., 2022). Venkataramani et al. (2012) proposed an ALS method called SALSA. It identifies the approximate don't-cares based on the maximum error limit, and then converts the ALS problem into a traditional logic synthesis problem with don't-cares. Scarabottolo et al. (2018) proposed a technique called CC. It identifies the maximum portion of an exact circuit that can be removed to derive an approximate circuit satisfying the WCE constraint. Recently, Witschen et al. (2022) proposed an ALS method called MUSCAT. It formulates the ALS problem under maximum error constraint as a minimal unsatisfiable subset problem, whose solution corresponds to an optimal approximate circuit satisfying the error constraint.

Some ALS methods can handle both average and maximum error constraints (Shin and Gupta, 2011; Chandrasekharan et al., 2016). Shin and Gupta (2011) proposed to replace a signal in the circuit with a constant 0 or 1, which considers ER and WCE constraint. Chandrasekharan et al. (2016) proposed an ALS method based on rewriting of AIGs, which works under ER, WCE, and maximum hamming distance constraint.

Precise computation of average and maximum errors caused by a LAC requires an exhaustive evaluation of all input patterns, which is not scalable to large circuits. Therefore, it is necessary to develop efficient error estimation techniques.

A common strategy to estimate average errors is through *Monte Carlo* (*MC*) simulation (Venkatesan et al., 2011). Su et al. (2022) further improved the MC simulation method, and proposed VECBEE, a versatile efficiency-accuracy configurable batch error estimation method for greedy ALS. They utilized the change propagation matrix to capture whether a signal change will be propagated to each circuit output, and based on it, estimated the average error of each LAC efficiently.

However, MC methods are not applicable to maximum error estimation. Therefore, some special methods are proposed, which can be divided into two categories. The first category estimates an upper bound on the maximum error. For example, Schlachter et al. (2017) provided a WCE bound for each node by summing the significance of all reachable outputs of the node. Scarabottolo et al. (2021) proposed a partition and propagation technique that derives a tighter maximum error bound for each node. However, both works (Schlachter et al., 2017; Scarabottolo et al., 2021) only consider a simple LAC proposed in the reference (Shin and Gupta, 2011), that is, replacing a signal by a constant 0 or 1. They do not support more complex LACs, such as the ones in the references (Venkataramani et al., 2013; Wu and Qian, 2016; Meng et al., 2020), which can further simplify the circuit and reduce more area, power, and delay. To estimate the maximum error for complex LACs, methods

belonging to the second category are proposed, which directly check whether the maximum error caused by each LAC exceeds the error bound or not. For instance, Venkatesan et al. (2011) proposed an exact maximum error checking method based on SAT. They construct a virtual error circuit to check whether the maximum error is satisfied or not, which will be converted into a SAT problem and solved by a SAT solver. In the references (Chandrasekharan et al., 2016) and (Češka et al., 2017), the exact maximum error checking method proposed in the reference (Venkatesan et al., 2011) is utilized to check the maximum error caused by each LAC in a circuit. However, since there are an enormous number of LACs, the SAT solver will be repeatedly called for many times, and hence, the exact maximum error checking method is time-consuming.

The work of this chapter belongs to the second category of maximum error estimation methods. As mentioned above, there are two important issues for previous maximum error estimation methods. First, the method estimating an upper bound does not support complex LACs. Second, the exact maximum error checking method is time-consuming. To address both issues, an efficient maximum error checking method supporting complex LACs is designed.

The main contributions of this chapter are as follows:

- This chapter proposes MECALS, a maximum error checking technique for ALS, which supports more complex LACs.
- MECALS models the maximum error caused by applying LACs with *partial Boolean difference* (*PBD*) and performs fast error checking using SAT sweeping.
- In MECALS, both exact and approximate ways of computing PBDs are proposed. This chapter also combines exact and approximate PBDs to tune the accuracy of MECALS.
- Based on MECALS, an efficient ALS flow is designed, which reduces circuit area and delay significantly.

The experimental results show that compared with the state-of-the-art ALS method, the MECALS-based flow accelerates by 13×, and further reduces circuit area and delay by 13.4% and 6.4%, respectively.

The rest of the chapter is organized as follows. Section 5.2 introduces the preliminaries. Section 5.3 elaborates the methodology of MECALS. Section 5.4 describes the ALS flow based on MECALS. Section 5.5 presents the experimental results. Finally, Section 5.6 summarizes the chapter.

5.2 Preliminaries

This section presents the preliminaries related to this chapter. For the circuit terminologies, please refer to Section 2.2. Next, *partial Boolean difference* (*PBD*) and maximum error metrics wil be introduced as follows.

5.2.1 Partial Boolean Difference

PBD is an important tool to diagnose errors in circuits (Chiang et al., 1972). The first-order PBD of a function $f = f(n_1, n_2, \ldots, n_M)$ *w.r.t.* one of its variables, n_i, is defined as

$$\begin{aligned} \Delta_{n_i} f &= f_{n_i} \oplus f_{\overline{n_i}} \\ &= f(n_1, \ldots, n_i = 1, \ldots, n_M) \oplus f(n_1, \ldots, n_i = 0, \ldots, n_M), \end{aligned} \tag{5.1}$$

where f_{n_i} and $f_{\overline{n_i}}$ are the *positive* and *negative cofactors* of f *w.r.t.* n_i, respectively, which are obtained by setting n_i in f to 1 and 0, respectively. The PBD $\Delta_{n_i} f$ compares f_{n_i} and $f_{\overline{n_i}}$ with the XOR function, and in this way, captures how f changes with n_i. If f_{n_i} and $f_{\overline{n_i}}$ are different under an input combination on $n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_M$, then $\Delta_{n_i} f = 1$, implying that f changes after flipping the value of n_i under the input combination. Otherwise, f is not affected by n_i under the input combination. Besides Eq. (5.1), another way to calculate the first-order PBD $\Delta_{n_i} f$ is as follows (Chiang et al., 1972):

$$\begin{aligned} \Delta_n f &= f(n_1, \ldots, n, \ldots, n_M) \oplus f(n_1, \ldots, \overline{n}, \ldots, n_M) \\ &= f \oplus f(n_1, \ldots, \overline{n}, \ldots, n_M). \end{aligned} \tag{5.2}$$

It should be noted that for the PBD $\Delta_{n_i} f$ in a circuit, n_i can represent any node, such as a PI, a PO, or an internal node, while f can be n_i or any TFO of n_i.

To capture the influence of two different variables n_i and n_j on f, the second-order PBD *w.r.t.* n_i and n_j is defined as

$$\Delta^2_{n_i n_j} f = \Delta_{n_j} (\Delta_{n_i} f) = f_{\overline{n_i}\,\overline{n_j}} \oplus f_{\overline{n_i}\, n_j} \oplus f_{n_i\, \overline{n_j}} \oplus f_{n_i n_j}, \tag{5.3}$$

where $f_{\overline{n_i}\,\overline{n_j}}$, $f_{\overline{n_i}\, n_j}$, $f_{n_i\, \overline{n_j}}$, and $f_{n_i n_j}$ are cofactors of f *w.r.t.* n_i and n_j, which are obtained by setting $n_i n_j$ to 00, 01, 10, and 11, respectively. The second-order PBD evaluates the overall influence of n_i and n_j on f by accumulating these cofactors together with the XOR function.

More generally, the kth-order ($k \geq 2$) PBD *w.r.t.* k different variables $n_{i_1}, \ldots, n_{i_k}$ ($1 \leq i_1, \ldots, i_k \leq M$) is

defined as

$$\Delta^{k}_{n_{i_1}\ldots n_{i_k}} f = \Delta_{n_{i_k}} \left(\Delta^{k-1}_{n_{i_1}\ldots n_{i_{k-1}}} f \right). \tag{5.4}$$

We can see that the kth-order PBD is derived from the $(k-1)$th-order PBD. Similarly, the kth-order PBD measures the overall influence of the k variables $n_{i_1}, \ldots, n_{i_k}$ on f.

In the rest of the chapter, PBD refers to the first-order PBD unless otherwise specified.

5.2.2 Maximum Error Metrics

This chapter focuses on an essential error metric of approximate circuits, the maximum error. Let $\boldsymbol{y} : \mathbb{B}^I \to \mathbb{B}^O$ and $\hat{\boldsymbol{y}} : \mathbb{B}^I \to \mathbb{B}^O$ be the multiple-output Boolean functions of an exact and an approximate circuit, respectively. The maximum error is defined as the maximum deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ over all possible input patterns as follows,

$$\text{maximum error} = \max_{\boldsymbol{x} \in \mathbb{B}^I} \varepsilon(\boldsymbol{y}(\boldsymbol{x}), \hat{\boldsymbol{y}}(\boldsymbol{x})), \tag{5.5}$$

where $\boldsymbol{y}(\boldsymbol{x})$ and $\hat{\boldsymbol{y}}(\boldsymbol{x})$ denote the outputs of the exact and the approximate circuits under the input pattern x, respectively and ε is a distance function measuring the deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. For example, for WCE, $\varepsilon(\boldsymbol{y}, \hat{\boldsymbol{y}}) = |int(\boldsymbol{y}) - int(\hat{\boldsymbol{y}})|$, where the function $int(v)$ returns the integer encoded by the binary vector v. For MaxSE, $\varepsilon(\boldsymbol{y}, \hat{\boldsymbol{y}}) = [int(\boldsymbol{y}) - int(\hat{\boldsymbol{y}})]^2$.

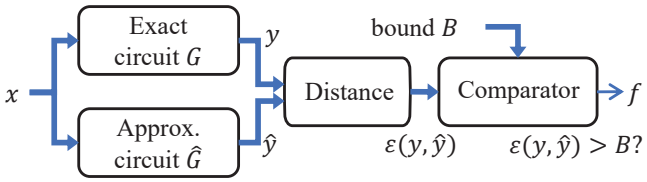


Figure 5.1: Virtual error circuit to check maximum error.

Maximum error can be checked by the *virtual error circuit* (Venkatesan et al., 2011) shown in Fig. 5.1, consisting of an exact circuit, an approximate circuit, a distance unit, and a comparator. The exact and approximate circuits take the same PIs, and their corresponding outputs are $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, respectively. The distance unit computes the distance between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, namely, $\varepsilon(\boldsymbol{y}, \hat{\boldsymbol{y}})$. The comparator checks whether the distance exceeds the maximum error bound B. If $\varepsilon(\boldsymbol{y}, \hat{\boldsymbol{y}}) > B$, then $f = 1$; otherwise, $f = 0$. In other words, if f is identically 0, *i.e.*, $f \equiv 0$, then the maximum error of $\hat{G}$ does not exceed B. Such a check is usually done by a SAT solver.

5.3 MECALS Methodology

This section presents the methodology of MECALS, a maximum error checking technique for ALS.

Given an exact circuit G and a maximum error bound B, an ALS flow generates an approximate circuit satisfying the bound. A widely-used ALS flow is the iterative ALS flow (Scarabottolo et al., 2020), which simplifies the circuit iteratively. For each iteration, there is an *intermediate approximate circuit* $\hat{G}$ satisfying the error constraint. To further simplify $\hat{G}$, many candidate LACs are generated. However, some candidates may not satisfy the given maximum error bound. Thus, the next step in the flow is to check whether *each* candidate LAC satisfies the given bound, *i.e.*, the LAC is *valid*. Finally, one valid LAC is selected based on some criteria and applied to simplify $\hat{G}$. Due to an enormous number of candidate LACs, *efficiently checking the validness of all the LACs* is a challenging task, and this is the problem MECALS tries to solve.

The following sections present the proposed solution, MECALS. First, Section 5.3.1 introduces a theoretical foundation of MECALS, which is a necessary and sufficient condition for a LAC to be valid. Then, Section 5.3.2 presents the details of MECALS, that is, how to check the validness of all candidate LACs efficiently. MECALS builds upon an important component, which is the construction of a circuit for calculating various PBDs. Its detail is described in Section 5.3.3.

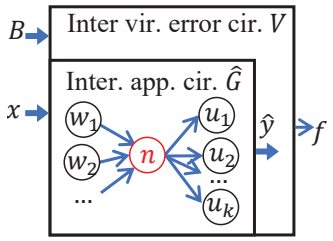
5.3.1 A Necessary and Sufficient Condition for a LAC to be Valid

This section presents a necessary and sufficient condition for a LAC to be valid, which is an important theoretical foundation of MECALS. A type of widely-used LACs, the *single-output LAC* (Shin and Gupta, 2011; Venkataramani et al., 2013; Wu and Qian, 2016; Wu et al., 2017; Liu and Zhang, 2017; Meng et al., 2020; Scarabottolo et al., 2018; Schlachter et al., 2017; Witschen et al., 2022; Chandrasekharan et al., 2016), is studied. It introduces approximation by modifying a single-output local circuit.

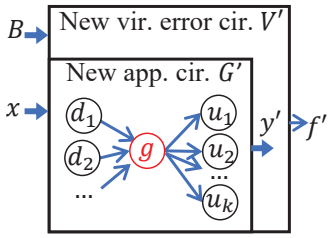
To introduce the condition, let us start from the intermediate virtual error circuit V in Fig. 5.2(a), which checks whether the intermediate approximate circuit $\hat{G}$ satisfies the given maximum error bound. V's PO f represents whether the maximum error of $\hat{G}$ exceeds the error bound B or not. Since $\hat{G}$ satisfies the maximum error constraint according to its definition above, we have $f \equiv 0$ according to the property of the virtual error circuit described above.

Assume that the only output of the local circuit on which a single-output LAC is applied is n, and after applying the LAC, the new output node of the local circuit is g. Correspondingly, the virtual error circuit in Fig. 5.2(a) changes to a new one shown in Fig. 5.2(b) with the PO f'. Checking whether the LAC is valid, *i.e.*, whether the error constraint is still satisfied after the LAC is applied, is equivalent to checking whether $f' \equiv 0$.

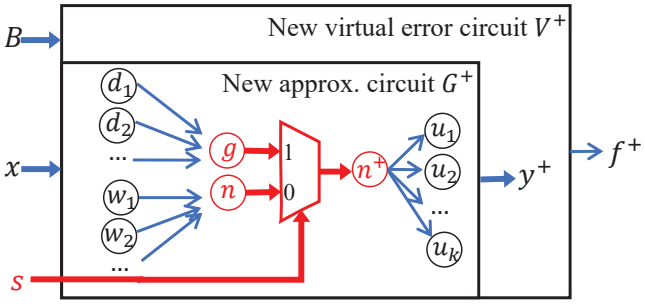
Note that the single-output LAC can be represented by a multiplexer. As shown in Fig. 5.2(c), the multiplexer takes g, n, and a control signal s as inputs, and it outputs a node n^+. If the control signal s is 1, then the LAC is applied, and n^+ equals g. Otherwise, the LAC is not applied, and n^+ equals n. Initially, the LAC is not applied, so $s = 0$.



(a) Intermediate virtual error circuit for the intermediate approximate circuit $\hat{G}$. n is the node to be approximated.



(b) New virtual error circuit for the new approximate circuit G', which is obtained by replacing n in $\hat{G}$ in Fig. 5.2(a) by g in Fig. 5.2(b).



(c) New virtual error circuit for the new approximate circuit G^+, which is obtained by replacing n in $\hat{G}$ in Fig. 5.2(a) by the multiplexer structure in Fig. 5.2(a).

Figure 5.2: Virtual error circuits used for maximum error checking. The maximum error bound is B.

The multiplexer representation helps check the validness of a LAC. After replacing the node n in V in Fig. 5.2(a) by the proposed multiplexer structure, another new virtual error circuit V^+ in Fig. 5.2(c), whose PO is f^+, can be obtained. By the above discussion, if $s = 0$, then $f^+ = f$. By the definition of cofactor, we further have

$$f = f_{\bar{s}}^+. \tag{5.6}$$

Similarly, we have

$$f' = f_s^+. \tag{5.7}$$

Since we have $f \equiv 0$, thus, in order to check whether $f' \equiv 0$, we only need to check whether $f \oplus f' \equiv 0$. By

the definition of PBD and Eqs. (5.6) and (5.7), it is equivalent to check whether

$$\Delta_s f^+ = f^+_{\bar{s}} \oplus f^+_s \equiv 0. \tag{5.8}$$

If so, the LAC is valid. Next, how to calculate $\Delta_s f^+$ is discussed.

Since every path from s to f^+ passes n^+, the PBD $\Delta_s f^+$ can be computed by the *chain rule* (Chiang et al., 1972):

$$\Delta_s f^+ = \Delta_s n^+ \Delta_{n^+} f^+, \tag{5.9}$$

where $\Delta_s n^+$ is the PBD of n^+ *w.r.t.* s, and $\Delta_{n^+} f^+$ is the PBD of f^+ *w.r.t.* n^+.

By the function of the multiplexer, we have $n^+ = \bar{s}n + sg$. Thus, by Eq. (5.1), we have $\Delta_s n^+ = n \oplus g$. Besides, n in $\hat{G}$ in Fig. 5.2(a) and n^+ in G^+ in Fig. 5.2(c) have the same set of TFOs, implying that f_n and $f^+_{n^+}$ are identical, and so are $f_{\bar{n}}$ and $f^+_{\overline{n^+}}$. Thus, we have $\Delta_{n^+} f^+ = \Delta_n f$ by Eq. (5.1). Therefore, Eq. (5.9) can be written as

$$\Delta_s f^+ = (n \oplus g)\Delta_n f. \tag{5.10}$$

Based on Eqs. (5.8) and (5.10), finally, the following claim can be reached, which states a necessary and sufficient condition for a LAC to be valid.

Theorem 5.1. *A LAC that replaces n by g is valid if and only if $(n \oplus g)\Delta_n f \equiv 0$.*

5.3.2 Details of MECALS

This section presents the details of MECALS, which tries to check the validness of all the LACs efficiently.

For a better elaboration, some notations are introduced first. Assume that the nodes in the intermediate approximate circuit $\hat{G}$ are $n_1, n_2, \ldots, n_N$. For each node n_i $(1 \le i \le N)$, assume that the number of available single-output LACs for n_i is L_i and that they replace the old node n_i by the new ones $g_{i1}, g_{i2}, \ldots, g_{iL_i}$, respectively. The node g_{ij} is constructed by r_{ij} divisors $d_{ij1}, d_{ij2}, \ldots, d_{ijr_{ij}}$ in $\hat{G}$, namely, $g_{ij} = \phi_{ij}(d_{ij1}, d_{ij2}, \ldots, d_{ijr_{ij}})$, where a divisor is an existing node in $\hat{G}$ and ϕ_{ij} is the corresponding Boolean function of g_{ij} *w.r.t.* the divisors. For simplicity, a LAC with the output node g_{ij} $(1 \le i \le N, 1 \le j \le L_i)$ is referred to as LAC g_{ij}. By Theorem 5.1, to check whether each LAC g_{ij} is valid, it is equivalent to checking whether $(n_i \oplus g_{ij})\Delta_n f \equiv 0$.

To check whether $(n_i \oplus g_{ij})\Delta_n f \equiv 0$ for all the candidate LACs g_{ij}, MECALS first builds a *maximum error checking circuit* shown in Fig. 5.3. It consists of an intermediate virtual error circuit V, a *PBD circuit* that computes the PBDs $\Delta_{n_i} f$'s, and peripheral circuits to derive the *validness signal* $h_{ij} = (n_i \oplus g_{ij})\,\Delta_{n_i} f$ for each

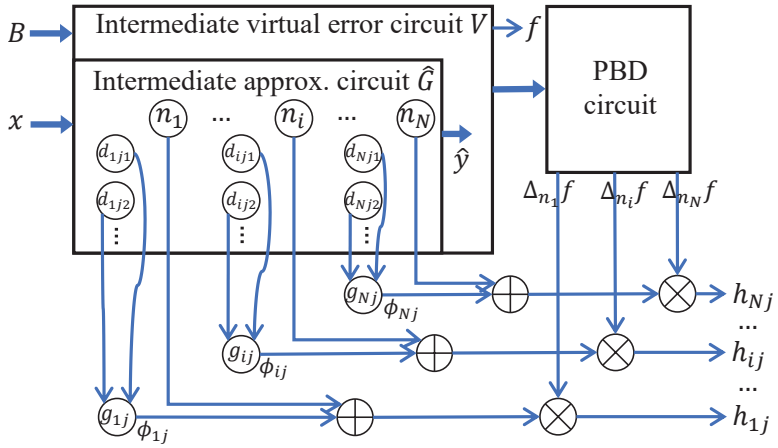


Figure 5.3: Maximum error checking circuit for all LAC g_{ij}'s in the intermediate approximate circuit $\hat{G}$. The LAC g_{ij} replaces the node n_i with a new node $g_{ij} = \phi_{ij}(d_{ij1}, d_{ij2}, \ldots)$. The PBD circuit computes the PBD $\Delta_{n_i}f$. The POs are the validness signals h_{ij}'s, each of which represents the validness of the LAC g_{ij}. The symbols $\oplus$ and $\otimes$ denote XOR and AND, respectively.

LAC g_{ij}. If $h_{ij} \equiv 0$, then the LAC g_{ij} is valid; otherwise, g_{ij} is invalid. The PBD circuit is an important component in MECALS, which will be described in detail in Section 5.3.3.

MECALS then applies SAT sweeping (Zhu et al., 2006) to the maximum error checking circuit to efficiently check the validness of all LACs in $\hat{G}$. SAT sweeping is a powerful SAT-based method that can detect functional equivalence in a circuit. Theoretically, after performing SAT sweeping on the maximum error checking circuit, all h_{ij}'s that are identically 0 will be found. If $h_{ij} \equiv 0$, then the corresponding LAC g_{ij} is valid. Otherwise, it is invalid. In reality, for the concern of efficiency, modern SAT sweeping methods, such as the one in the reference (Zhang et al., 2021), only detect most of the functional equivalence cases in a circuit, while the remaining small portion may be unidentified. Therefore, using SAT sweeping, MECALS can also find most of valid LACs whose h_{ij}'s are identically 0.

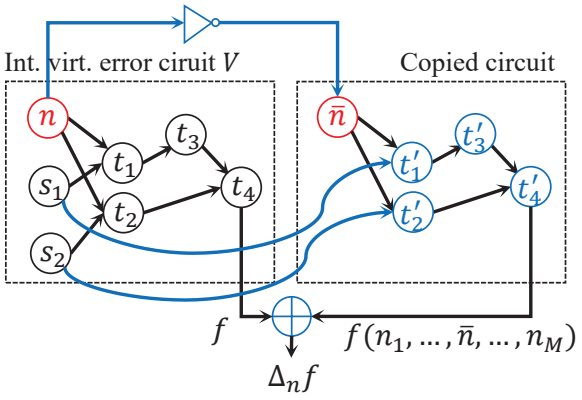


Figure 5.4: Circuit to compute the exact PBD $\Delta_n f$. The symbol $\oplus$ denotes XOR.

5.3.3 Construction of the PBD Circuit

This section describes how to construct the PBD circuit in Fig. 5.3, which computes the PBD $\Delta_{n_i} f$ for each node n_i in the intermediate approximate circuit $\hat{G}$. An exact and an approximate method to compute PBDs will be introduced. For simplicity, n_i is referred to as n in this section.

Exact PBD Circuit

This subsection describes how to build an exact PBD circuit. Note that in Eq. (5.2), f is just the PO node of the intermediate virtual error circuit V in Fig. 5.2(a). Thus, to compute $\Delta_n f$, we need to compute the unknown term $f(n_1, \ldots, \overline{n}, \ldots, n_M)$ in Eq. (5.2). As shown in Fig. 5.4, this term can be computed by copying nodes that are n's TFOs (*i.e.*, t_1, t_2, t_3, and t_4) in V and driving the copied nodes (*i.e.*, t_1', t_2', t_3', and t_4') with $\overline{n}$. The new PO node t_4' copied from the old PO node t_4 is just $f(n_1, \ldots, \overline{n}, \ldots, n_M)$. Then, $\Delta_n f$ is the XOR of f and $f(n_1, \ldots, \overline{n}, \ldots, n_M)$.

However, if the circuit V and the intermediate approximate circuit $\hat{G} \subset V$ have M and N nodes, respectively, then $O(MN)$ nodes are copied to construct the PBD circuit for all nodes in $\hat{G}$. As a result, for large benchmarks, the maximum error checking circuit is too large to be handled by SAT sweeping.

Approximate PBD Circuit

This subsection proposes how to build an approximate PBD circuit that computes approximate PBDs. First, the derivation of approximate PBDs is described. Then, how to build the approximate PBD circuit is introduced.

To derive the approximate PBD, let us start from a recursive formula of computing PBDs exactly proposed in Chiang et al. (1972). It calculates the PBD of n based on the PBDs of n's fanouts $u_1, u_2, \ldots, u_k$ $(k \geq 1)$ as follows:

$$\begin{aligned}\Delta_n f = &(\Delta_n u_1 \Delta_{u_1} f) \oplus (\Delta_n u_2 \Delta_{u_2} f) \oplus \cdots \oplus (\Delta_n u_k \Delta_{u_k} f) \\ &\oplus \left(\Delta_n u_1 \Delta_n u_2 \Delta^2_{u_1 u_2} f\right) \oplus \left(\Delta_n u_1 \Delta_n u_3 \Delta^2_{u_1 u_3} f\right) \oplus \cdots \\ &\oplus \left(\Delta_n u_{k-2} \Delta_n u_k \Delta^2_{u_{k-2} u_k} f\right) \oplus \left(\Delta_n u_{k-1} \Delta_n u_k \Delta^2_{u_{k-1} u_k} f\right) \\ &\oplus \cdots \oplus \left(\Delta_n u_1 \Delta_n u_2 \cdots \Delta_n u_k \Delta^k_{u_1 u_2 \ldots u_k} f\right),\end{aligned} \tag{5.11}$$

where $\Delta_n u_i (1 \leq i \leq k)$ is the first-order PBD of the fanout u_k *w.r.t.* n, which can be obtained directly by Eq. (5.1), $\Delta_{u_i} f (1 \leq i \leq k)$ is the first-order PBD of the output f *w.r.t.* the fanout u_i, $\Delta^2_{u_i u_j} f (i \neq j, 1 \leq i, j \leq k)$ is the second-order PBD of f *w.r.t.* u_i and u_j, and so on. The base case of the recursion is $\Delta_f f$, which equals

1 by definition.

Observing that the *right-hand side* (*RHS*) of Eq. (5.11) contains terms $\Delta_n u_1, \ldots, \Delta_n u_k$, we can rewrite Eq. (5.11) by Shannon's expansion *w.r.t.* $\Delta_n u_1, \ldots, \Delta_n u_k$ as

$$\begin{aligned}
\Delta_n f = {} & \overline{\Delta_n u_1}\,\overline{\Delta_n u_2}\cdots\overline{\Delta_n u_k}\,\Delta_n f(\Delta_n u_1 = 0, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0) \\
& + \Delta_n u_1\,\overline{\Delta_n u_2}\cdots\overline{\Delta_n u_k}\,\Delta_n f(\Delta_n u_1 = 1, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0) \\
& + \overline{\Delta_n u_1}\,\Delta_n u_2\cdots\overline{\Delta_n u_k}\,\Delta_n f(\Delta_n u_1 = 0, \Delta_n u_2 = 1, \ldots, \Delta_n u_k = 0) \\
& + \Delta_n u_1 \Delta_n u_2\cdots\overline{\Delta_n u_k}\,\Delta_n f(\Delta_n u_1 = 1, \Delta_n u_2 = 1, \ldots, \Delta_n u_k = 0) \\
& + \cdots + \Delta_n u_1 \Delta_n u_2 \cdots \Delta_n u_k \Delta_n f(\Delta_n u_1 = 1, \Delta_n u_2 = 1, \ldots, \Delta_n u_k = 1),
\end{aligned} \tag{5.12}$$

where $\overline{\Delta_n u_1}\,\overline{\Delta_n u_2}\cdots\overline{\Delta_n u_k}$, $\Delta_n u_1\,\overline{\Delta_n u_2}\cdots\overline{\Delta_n u_k}$, and so on are 2^k *Shannon factors*, and $\Delta_n f(\Delta_n u_1 = 0, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0)$, $\Delta_n f(\Delta_n u_1 = 1, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0)$, and so on are the corresponding *Shannon cofactors*.

To obtain the Shannon cofactor $\Delta_n f(\Delta_n u_1 = 0, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0)$, we substitute all $\Delta_n u_i$'s $(1 \le i \le k)$ in the RHS of Eq. (5.11) with 0. After simplification, we obtain $\Delta_n f(\Delta_n u_1 = 0, \Delta_n u_2 = 0, \ldots, \Delta_n u_k = 0) = 0$. Similarly, we can obtain that for any $1 \le i \le k$, the Shannon cofactor

$$\begin{aligned}
&\Delta_n f(\Delta_n u_1 = 0, \ldots, \Delta_n u_{i-1} = 0, \Delta_n u_i = 1, \\
&\Delta_n u_{i+1} = 0, \ldots, \Delta_n u_k = 0) = \Delta_{n_i} f.
\end{aligned}$$

Thus, Eq. (5.12) can be reorganized as

$$\Delta_n f = \alpha \cdot 0 + \beta_1 \Delta_{u_1} f + \beta_2 \Delta_{u_2} f + \cdots + \beta_k \Delta_{u_k} f + \gamma, \tag{5.13}$$

where $\alpha = \overline{\Delta_n u_1}\,\overline{\Delta_n u_2}\cdots\overline{\Delta_n u_{k-1}}\,\overline{\Delta_n u_k}$ and $\beta_i = \overline{\Delta_n u_1}\,\overline{\Delta_n u_2}\cdots\ \Delta_n u_i \cdots \overline{\Delta_n u_{k-1}}\,\overline{\Delta_n u_k}$ $(1 \le i \le k)$ are the Shannon factors with 0 and 1 positive literal (*i.e.*, $\Delta_n u_i$), respectively. γ consists of the terms in Eq. (5.12) whose Shannon factors are neither α nor β_i.

Note that for each term in γ, its Shannon factor has more than 1 positive literal, and the corresponding Shannon cofactor contains higher-order (*i.e.*, non-first-order) PBDs. For example, for the Shannon factor $\Delta_n u_1 \Delta_n u_2\,\overline{\Delta_n u_3}\cdots\overline{\Delta_n u_k}$ with 2 positive literals $\Delta_n u_1$ and $\Delta_n u_2$, its corresponding Shannon cofactor is $\Delta_n f(\Delta_n u_1 = 1, \Delta_n u_2 = 1, \Delta_n u_3 = 0, \ldots, \Delta_n u_k = 0) = \Delta_{u_1} f \oplus \Delta_{u_2} f \oplus \Delta^2_{u_1 u_2} f$ (from Eq. (5.11)), which contains the second-order PBD $\Delta^2_{u_1 u_2} f$. Since the higher-order PBDs contained in the Shannon cofactors in γ

are difficult to derive, it is proposed to approximate them by setting all the Shannon cofactors in γ as 1. In this way, γ is simplified into a disjunction of all Shannon factors except for α and β_i, which can be written as $\gamma \approx \overline{\alpha + \beta_1 + \beta_2 + \cdots + \beta_k}$. Based on this, Eq. (5.13) is approximated as an approximate PBD, which is defined recursively as follows:

$$\widehat{\Delta}_n f = \beta_1 \widehat{\Delta}_{u_1} f + \beta_2 \widehat{\Delta}_{u_2} f + \cdots + \beta_k \widehat{\Delta}_{u_k} f + \overline{\alpha + \beta_1 + \beta_2 + \cdots + \beta_k}, \tag{5.14}$$

where $\widehat{\Delta}_n f$ is the approximate PBD of f *w.r.t.* n, and $\widehat{\Delta}_{u_i} f$ $(1 \leq i \leq k)$ is the approximate PBD of f *w.r.t.* the fanout u_i. For the special case where $n = f$, $\widehat{\Delta}_f f = 1$ is defined.

Although $\widehat{\Delta}_n f$ is an approximation to $\Delta_n f$, the following claim on it can be obtained, which indicates that $\widehat{\Delta}_n f$ is a conservative estimation of $\Delta_n f$.

Theorem 5.2. *For any node n in the intermediate virtual error circuit V, $\widehat{\Delta}_n f = 0$ implies $\Delta_n f = 0$, i.e., $\widehat{\Delta}_n f = 0 \Rightarrow \Delta_n f = 0$.*

Proof. The claim can be proved by mathematical induction following a reverse topological order of the nodes in the intermediate virtual error circuit V. The base case is $n = f$. In this case, since $\widehat{\Delta}_f f = 1$ by definition, the statement $\widehat{\Delta}_f f = 0$ is always false, which implies any statement. Thus, $\widehat{\Delta}_f f = 0 \Rightarrow \Delta_f f = 0$.

Now consider a node n, whose fanouts are $u_1, u_2, \ldots, u_k$. Assume that the claim holds for $u_1, u_2, \ldots, u_k$, that is, $\widehat{\Delta}_{u_i} f = 0 \Rightarrow \Delta_{u_i} f = 0$ for all $1 \leq i \leq k$. We want to prove $\widehat{\Delta}_n f = 0 \Rightarrow \Delta_n f = 0$.

Since $\widehat{\Delta}_n f = 0$, by Eq. (5.14), we have $\beta_i \widehat{\Delta}_{u_i} f = 0$ and $\alpha + \beta_1 + \beta_2 + \cdots + \beta_k = 1$.

First, we prove that $\beta_i \widehat{\Delta}_{u_i} f = 0 \Rightarrow \beta_i \Delta_{u_i} f = 0$ by discussing two possible cases causing $\beta_i \widehat{\Delta}_{u_i} f = 0$: 1) $\beta_i = 0$ and 2) $\widehat{\Delta}_{u_i} f = 0$. If $\beta_i = 0$, then apparently $\beta_i \Delta_{u_i} f = 0$. If $\widehat{\Delta}_{u_i} f = 0$, then by the assumption that $\widehat{\Delta}_{u_i} f = 0 \Rightarrow \Delta_{u_i} f = 0$, we have $\Delta_{u_i} f = 0$, and hence, $\beta_i \Delta_{u_i} f = 0$.

Then, we prove that $\alpha + \beta_1 + \beta_2 + \cdots + \beta_k = 1 \Rightarrow \gamma = 0$. If $\alpha + \beta_1 + \beta_2 + \cdots + \beta_k = 1$, then $\alpha = 1$ or $\beta_i = 1$. Recall that γ is a disjunction of all the terms in Eq. (5.12) with Shannon factors that are neither α nor β_i. Thus, given that either $\alpha = 1$ or $\beta_i = 1$, the Shannon factor of each term in γ is 0, and hence, $\gamma = 0$.

Based on the above discussion, it can be concluded that for the particular node n, if $\widehat{\Delta}_n f = 0$, then all terms in the RHS of Eq. (5.13) will be 0 and hence, $\Delta_n f = 0$.

Thus, by mathematical induction, we prove that for any node $n \in V$, $\widehat{\Delta}_n f = 0$ implies $\Delta_n f = 0$. □

By the above theorem, the following important corollary is conducted.

Corollary 5.1. *For any LAC in the intermediate approximate circuit $\hat{G}$ that replaces node n by node g, $(n \oplus g)\widehat{\Delta}_n f \equiv 0$ implies $(n \oplus g)\Delta_n f \equiv 0$, i.e., $(n \oplus g)\widehat{\Delta}_n f \equiv 0 \Rightarrow (n \oplus g)\Delta_n f \equiv 0$.*

Proof. We first prove $(n \oplus g)\widehat{\Delta}_n f = 0 \Rightarrow (n \oplus g)\Delta_n f = 0$ under a certain input pattern of $\hat{G}$. If $(n \oplus g)\widehat{\Delta}_n f = 0$, there are two possible cases: 1) $n \oplus g = 0$ and 2) $\widehat{\Delta}_n f = 0$. If $n \oplus g = 0$, then obviously $(n \oplus g)\Delta_n f = 0$. If $\widehat{\Delta}_n f = 0$, then by Theorem 5.2, we have $\Delta_n f = 0$ and hence, $(n \oplus g)\Delta_n f = 0$.

Therefore, if $(n \oplus g)\widehat{\Delta}_n f$ is identically 0, which means that $(n \oplus g)\widehat{\Delta}_n f = 0$ for each input pattern, then we have $(n \oplus g)\Delta_n f = 0$ for each input pattern, indicating that $(n \oplus g)\Delta_n f$ is identically 0. □

Based on Corollary 5.1, it is feasible to *approximately* check the validness of each LAC by checking whether $(n \oplus g)\widehat{\Delta}_n f \equiv 0$. If it is true, then $(n \oplus g)\Delta_n f \equiv 0$, indicating that the LAC is valid. Otherwise, it is uncertain whether $(n \oplus g)\Delta_n f$ is identically 0, and conservatively, the LAC is treated as invalid.

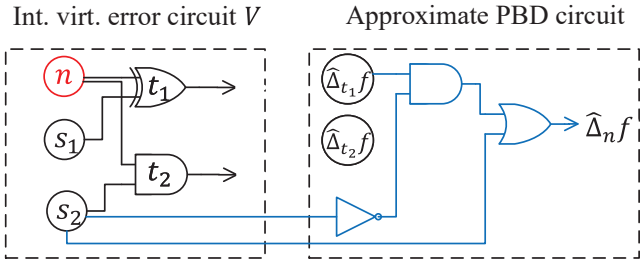


Figure 5.5: Circuit to compute the approximate PBD $\widehat{\Delta}_n f$.

After presenting theories of approximate PBDs, how to construct an approximate PBD circuit is explained. At the beginning, the node that computes $\widehat{\Delta}_f f$ is initialized as a constant 1. Then, by the recursive formula shown in Eq. (5.14), each node n in the intermediate virtual error circuit V can be traversed following a reverse topological order, and construct the corresponding node that computes $\widehat{\Delta}_n f$. Fig. 5.5 gives an example to build a circuit to compute $\widehat{\Delta}_n f$. Assume n has two fanouts t_1 and t_2, whose functions are $t_1 = n \oplus s_1$ and $t_2 = ns_2$, respectively. By Eq. (5.14), we have $\widehat{\Delta}_n f = \Delta_n t_1 \overline{\Delta_n t_2}\, \widehat{\Delta}_{t_1} f + \overline{\Delta_n t_1}\, \Delta_n t_2 \widehat{\Delta}_{t_2} f + \Delta_n t_1 \Delta_n t_2$. Since $\Delta_n t_1 = 1$ and $\Delta_n t_2 = s_2$ by Eq. (5.1), we further have $\widehat{\Delta}_n f = \overline{s_2}\, \widehat{\Delta}_{t_1} f + s_2$, which can be implemented by the circuit shown in Fig. 5.5. Compared with the exact PBDs, computation of approximate PBDs does not copy n's TFOs. Instead, the approximate PBD of a node n depends on the approximate PBDs of n's fanouts. In this way, the approximate PBD circuit has much smaller size, which is applicable on larger circuits.

Combining Exact and Approximate PBDs

This subsection proposes to build a circuit that combines exact and approximate PBDs. Generally speaking, exact PBDs are computed for nodes with more fanouts, while approximate PBDs are computed for nodes with fewer fanouts. The reason is that more fanouts mean more terms with higher-order PBDs discarded during the approximation from Eq. (5.13) to Eq. (5.14). In this case, the approximate PBDs for nodes with many fanouts

suffer from a large error, compared to the exact PBD. Therefore, for nodes with more fanouts, exact PBDs are computed to guarantee their accuracy, and for nodes with fewer fanouts, approximate PBDs are used for the sake of efficiency.

In practice, combined PBDs for each node in the intermediate approximate circuit $\hat{G}$ can be computed as follows. Initially, for each output of $\hat{G}$, the exact PBD is computed. Then, it is determined which internal (*i.e.*, non-input and non-output) nodes in $\hat{G}$ utilize exact PBDs, and which ones utilize approximate PBDs. To tune the accuracy of combined PBDs, a parameter P is defined as the percentage of nodes using exact PBDs. After sorting the internal nodes in $\hat{G}$ by the fanout number in the descending order, the first $P\%$ nodes with more fanouts utilize exact PBDs and the rest $(100 - P)\%$ nodes with fewer fanouts use approximate PBDs. Finally, the combined PBD for each internal node of $\hat{G}$ can be updated in reverse topological order.

5.4 ALS Flow Based on MECALS

Algorithm 11: MECALS-based ALS procedure.

Input: exact circuit G, maximum error bound B, percentage of nodes using exact PBDs P.
Output: approximate circuit $\hat{G}$.

```
1  Intermediate approximate circuit Ĝ ← G;
2  while True do
3      V ← BuildVirtualErrorCircuit(G, Ĝ, B);
4      G_p ← BuildPBDCircuit(Ĝ, V, P);
5      Determine the set of LACs {g_ij} in Ĝ;
6      G_e ← BuildErrorCheckCircuit(V, G_p, {g_ij});
7      SATSweeping(G_e);
8      foreach LAC g_ij ∈ {g_ij} do
9          h_ij ← validness signal of g_ij in G_e;
10         if h_ij ≡ 0 then ApplyLAC(Ĝ, g_ij); break;
11     if Ĝ is not modified then break;
12 return Ĝ
```

This section presents an ALS flow based on MECALS, which is shown in Algorithm 11. The inputs are an exact circuit G, a maximum error bound B, and a percentage of nodes using exact PBDs, P. Its output is an approximate circuit $\hat{G}$ that satisfies the error constraint. The flow initializes the intermediate approximate circuit $\hat{G}$ with the exact circuit G (Line 1), and then approximates the circuit iteratively. For each iteration, an intermediate virtual error circuit V (Line 3) shown in Fig. 5.2(a) is built. Based on the circuit V, a PBD circuit G_p is built to compute combined PBDs with the method introduced in Section 5.3.3 (Line 4). Then, the set of LACs $\{g_{ij}\}$ in $\hat{G}$ are determined (Line 5). Next, a maximum error checking circuit G_e (Line 6) shown in Fig. 5.3

is built, and perform SAT sweeping on G_e (Line 7). After SAT sweeping, it enumerates each LAC g_{ij} (Line 8) and obtains the corresponding validness signal h_{ij} in G_e (Line 9). Simpler LACs such as the constant LAC (Shin and Gupta, 2011) are visited first, and more complex LACs such as the SASIMI LAC (Venkataramani et al., 2013) are visited later. For each LAC, its validness is tested by checking whether $h_{ij} \equiv 0$ (Line 10). If it is true, then g_{ij} is applied into $\hat{G}$ and the rest LACs are skipped. The flow terminates until the approximate circuit $\hat{G}$ is not modified, which means there are no valid LACs satisfying the error constraint (Line 11). Finally, the approximate circuit $\hat{G}$ is returned (Line 12).

5.5 Results

This section presents the experimental results of MECALS.

5.5.1 Experiment Setup

MECALS and the MECALS-based ALS flow are implemented in C++ and tested on a single core of an AMD 4800H processor with 32GB RAM. Currently, the proposed flow works on AIGs, although it also supports other circuit representations. The proposed flow integrates the ABC (Mishchenko et al., 2022) software, which is a state-of-the-art logic synthesis and verification system. To evaluate the hardware cost, circuits are mapped into the Nangate 45nm library (Nangate, Inc., 2022) with the ABC command "amap", and the area and delay of the mapped circuits are measured by the ABC command "stime". To implement the SAT-sweeping method mentioned in Section 5.3.2, the ABC command "ifraig" is utilized. The proposed flow considers two kinds of single-output LACs, the constant LAC (Shin and Gupta, 2011) and the SASIMI LAC (Venkataramani et al., 2013). Benchmarks used in the experiments are listed in Table 5.1, in which circuit names, PI/PO numbers, gate numbers, circuit area, and circuit delay are shown in each column. The first five benchmarks are the ones used in MUSCAT (Witschen et al., 2022), one of the state-of-the-art ALS methods under the maximum error constraint proposed recently, and the last three benchmarks are the arithmetic circuits from the BACS benchmark (Scarabottolo et al., 2020).

5.5.2 Accuracy-Efficiency of MECALS

This experiment shows the accuracy-efficiency tradeoff of MECALS. MECALS is compared with a SAT-based exact maximum error checking method (exact method in short), which is used in many previous works (Češka et al., 2017; Venkatesan et al., 2011; Chandrasekharan et al., 2016).

Table 5.1: Benchmarks used in experiments

Circuit	I/O	#Gate	Area/μm^2	Delay/ps
add8u	16/9	41	42.03	359.94
absdiff	16/8	80	87.25	416.28
rca32	64/33	190	184.60	1843.34
am8	16/16	422	435.44	1255.23
binsqrd	16/18	1047	1052.30	1526.81
mac	12/8	91	92.83	595.88
buttfly	32/34	187	170.51	1008.15
mult16	32/32	1500	1418.84	1981.54

Notably, the maximum error checking problem is inherently a binary classification problem that divides LACs in the intermediate approximate circuit $\hat{G}$ into two categories, namely, valid LACs and invalid LACs. Therefore, the accuracy of MECALS can be evaluated by two common metrics for binary classifiers, *precision* and *recall*. Their definitions are as follows:

$$\begin{aligned} \text{precision} &= \frac{TP}{TP+FP}, \\ \text{recall} &= \frac{TP}{TP+FN}, \end{aligned} \tag{5.15}$$

where TP (*true positive*) denotes the number of LACs that are checked as valid by both MECALS and the exact method, FP (*false positive*) denotes the number of LACs that are checked as valid by MECALS but invalid by the exact method, and FN (*false negative*) denotes the number of LACs that are checked as invalid by MECALS but valid by the exact method. Higher precision and recall are preferred, and a perfect classifier has precision and recall both equal to 1.

Considering the limited scalability of the exact method, only three small benchmarks *add8u*, *absdiff*, and *mac* are tested with the SASIMI LAC with a WCE bound 16. When counting the LACs used in TP, FP, and FN, only the LACs in the first three iterations of the proposed ALS flow are considered. Table 5.2 shows precision and recall of MECALS versus different P's, the percentages of nodes using exact PBDs (mentioned in Section 5.3.3). It is worth noting that FP is always 0 and precision is always 1.0 in all cases, indicating that no LACs are checked as valid by MECALS but invalid by the exact method. This guarantees that each valid LAC found by MECALS can be applied to further simplify the circuit without violating the error constraint. In addition, it is worth noting that as P increases, recall also increases for all benchmarks. When $P = 100$, namely, all nodes use exact PBDs, FN is very small and recall is equal to or very close to 1.0 for all benchmarks, indicating that few LACs are checked as invalid by MECALS but valid by the exact method. In other words, MECALS cannot detect all LACs that are checked as valid by the exact method. The reason is that the SAT

sweeping method "ifraig" does not detect all functional equivalence in the error checking circuit, as mentioned in Section 5.3.2.

Table 5.2: Precision and recall of MECALS over different percentages of nodes using exact PBDs P.

P		0	20	40	60	80	100
add8u	TP	2844	4249	4479	4479	4483	4485
	FP	0	0	0	0	0	0
	FN	1641	236	6	6	2	0
	Prec.	1.000	1.000	1.000	1.000	1.000	1.000
	Recall	0.634	0.947	0.999	0.999	1.000	1.000
absdiff	TP	12874	13933	14434	14659	14746	14872
	FP	0	0	0	0	0	0
	FN	1646	940	439	214	127	1
	Prec.	1.000	1.000	1.000	1.000	1.000	1.000
	Recall	0.887	0.937	0.970	0.986	0.991	1.000
mac	TP	5057	14778	19697	19758	19827	20031
	FP	0	0	0	0	0	0
	FN	18858	5571	652	591	522	318
	Prec.	1.000	1.000	1.000	1.000	1.000	1.000
	Recall	0.211	0.726	0.968	0.971	0.974	0.984

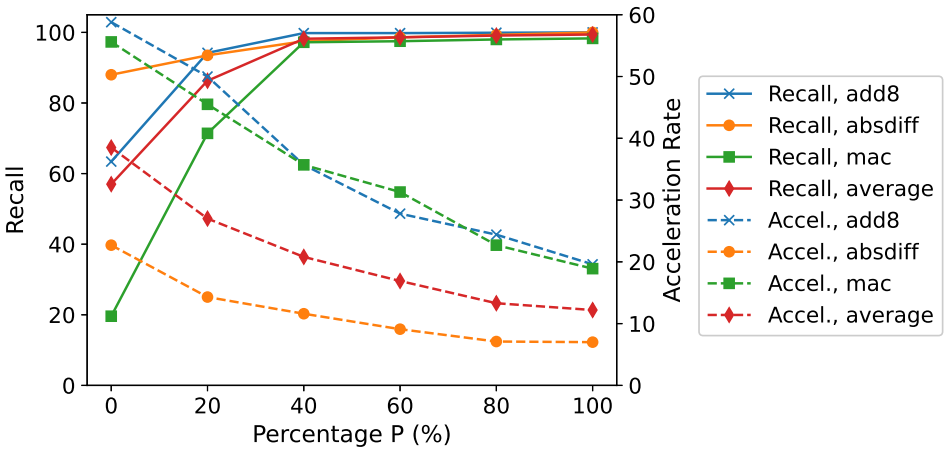


Figure 5.6: Accuracy-efficiency tradeoff of MECALS.

Furthermore, curves of recall and acceleration rate, defined as the runtime of the exact method over the runtime of MECALS, are plotted in Figure 5.6. It can be seen that as P increases, recall increases, but acceleration rate decreases. When P is no less than 40, the recall for each circuit is almost 100%, implying that MECALS can detect almost the same set of valid LACs as the exact method. Compared with the exact method, acceleration rate of MECALS ranges from 12× to 39× for different P's on average.

5.5.3 Comparison of MECALS-Based ALS Flow with MUSCAT

This experiment compares the proposed ALS flow based on MECALS with MUSCAT (Witschen et al., 2022), which is a state-of-the-art method proposed recently. The comparison is on the benchmarks in Table 5.1 under the WCE constraint.

The first five circuits are also reported in the MUSCAT paper, each of which is approximately synthesized under tens of different WCE bounds. The proposed ALS flow is tested under exactly the same WCE bounds as MUSCAT. Since there are many configurations in MUSCAT that produce approximate circuits of different quality, the best configuration from the MUSCAT paper is selected so that the largest area savings is achieved. The best configurations of *add8u*, *absdiff*, *rca32*, *binsqrd*, and *am8* are *must_5_1.0*, *must_30_1.0*, *must_20_1.0*, *z3_0.15*, and *must_7200_1.0*, respectively, whose concrete meanings can be found in the MUSCAT paper. For the last three circuits not reported in the MUSCAT paper, four different WCE bounds are chosen according to the PO number O, that is, $2^{\lfloor O/8 \rfloor}, 2^{\lfloor O/4 \rfloor}, 2^{\lfloor 3O/8 \rfloor}$ and $2^{\lfloor O/2 \rfloor}$, where $\lfloor\rfloor$ indicates rounding down. The configurations of *mac*, *buttfly*, and *mult16* are *z3_1.0*, *z3_1.0*, and *z3_0.01*, respectively. In the proposed ALS flow, different P's, *i.e.*, the percentages of nodes using exact PBDs (mentioned in Section 5.3.3), are chosen according to the circuit size. For the two largest circuits *binsqrd* and *mult16*, a smaller $P = 4$ is chosen to reduce runtime, while for the remaining ones, a larger $P = 40$ is chosen to improve the synthesis quality. Note that from Fig. 5.6, recall is almost 100% when $P = 40$, implying that MECALS can detect almost the same set of valid LACs as the exact maximum error checking method.

Table 5.3: Comparison of MECALS-based flow with MUSCAT. "N/A" means not able to generate approximate circuits in 24 hours.

Circuit	Mean area saving		Mean delay saving		Runtime/s	
	MUSCAT	MECALS	MUSCAT	MECALS	MUSCAT	MECALS
add8u	55.9%	**58.9%**	45.2%	**53.3%**	15	**1**
absdiff	47.0%	**67.6%**	26.6%	**42.5%**	67	**3**
rca32	40.3%	**47.7%**	45.8%	28.4%	29	611
binsqrd	9.8%	**27.0%**	5.5%	**8.2%**	31546	**791**
am8	61.5%	**75.8%**	38.7%	**53.3%**	7380	**1548**
mac	10.0%	**32.2%**	5.7%	**13.9%**	3	8
buttfly	15.1%	**24.0%**	4.4%	**17.7%**	8	51
Mean of the above seven	34.2%	**47.6%**	24.6%	**31.0%**	5578	**431**
mult16	N/A	14.7%	N/A	10.8%	>24h	**4526**

The *area saving* (the reduced area of the approximate circuit over the exact one) and the *delay saving* (the reduced delay of the approximate circuit over the exact one) are used to evaluate the approximate designs.

Apparently, larger area and delay savings are preferred. Average area savings, average delay savings, and average runtime are compared over all the different WCE bounds for each circuit. As shown in Table 5.3. MUSCAT cannot generate approximate designs for *mult16*, while the proposed method can deal with all benchmarks. Compared with MUSCAT on the first seven benchmarks, the proposed method saves area and delay by 47.6% and 31.0% on average, improving by 13.4% and 6.4% on average, respectively, and accelerates by 13× on average. Notably, the proposed method always reduces more area than MUSCAT, and reduces more delay on all benchmarks except for *rca32*. In addition, the proposed method is competitive on runtime, which is faster than MUSCAT on *add8u*, *absdiff*, *binsqrd*, and *am8*. In particular, for the large circuits *binsqrd* and *am8*, the proposed method accelerates by 40× and 5×, respectively. Furthermore, the proposed method is more scalable, which can synthesize *mult16* (with 1500 gates) in a reasonable time.

5.6 Summary

This chapter proposes MECALS, a maximum error checking technique for ALS. It relies on an error model with PBDs and fast error checking with SAT sweeping. Furthermore, an efficient ALS flow is designed based on MECALS. The experimental results show that the proposed flow improves the quality of ALS significantly, and accelerates ALS dramatically.


20006301

Chapter 6

DALTA: Decomposition-Based Approximate Lookup Table Architecture

This chapter proposes a high-quality approximate LUT architecture called DALTA. Moreover, efficient synthesis methods are developed to implement an arbitrary function with the DALTA architecture.

6.1 Motivations and Overview

As mentioned in Section 1.4.1, it is highly demanded to design high-quality approximate computing architectures. Therefore, in this chapter, the design of approximate LUT architectures is studied. Many existing approximate LUTs rely on the first-order Taylor expansion (Schulte and Stine, 1997; Muller, 1999; De Dinechin and Tisserand, 2005; Hsiao et al., 2016; Tian et al., 2017), which only supports the continuous functions and cannot be applied to arbitrary functions. To approximate arbitrary functions regardless of their continuity, this chapter proposes *DALTA*, a *decomposition-based approximate lookup table architecture*. The work in this chapter is inspired by the conventional Boolean function decomposition theory, which was pioneered by Ashenhurst (1959), Curtis (1962), and Roth and Karp (1962). Later, Lai et al. (1993) extended the theory using the BDD and presented how to decompose multiple-output functions with BDDs. Lee et al. (2008) further proposed to decompose large functions by interpolation and satisfiability solving. The work in this chapter extends the conventional function decomposition theory and proposes several approximate decomposition methods. With these methods, a function with many inputs can be approximately decomposed into two simpler functions with fewer inputs, which are stored in two smaller LUTs. In this way, the LUT size is cut down dramatically, and so is

the area, delay, and energy of the LUT architecture. Theoretically, the proposed method enjoys an exponential decrease in LUT size over the accurate LUT as the number of inputs increases.

The main contributions of this chapter are as follows:

- A decomposition-based approximate lookup table architecture called DALTA is proposed, which improves the area, delay, and energy dramatically by introducing a small error. DALTA can implement approximations for both continuous and non-continuous functions.

- To map a function into DALTA, specific synthesis algorithms are proposed, *i.e.*, approximate decomposition methods based on *integer linear programming* (*ILP*). Besides, an efficient heuristic method to derive approximate decomposition is designed.

DALTA is a high-speed energy-efficient architecture that is applicable to arbitrary functions. On the one hand, DALTA can calculate continuous functions with 56.5% energy saving and 92.4% latency reduction, compared with the state-of-the-art approximate LUT architecture, ApproxLUT (Tian et al., 2017). On the other hand, DALTA works well on non-continuous functions. Compared with the rounding-based approximate LUT architecture, it saves area, energy, and latency by 95.8%, 39.0%, and 98.3%, respectively.

The rest of the chapter is organized as follows. Section 6.2 provides the preliminaries on disjoint decomposition. Section 6.3 explains the basic idea. Section 6.4 elaborates DALTA. Section 6.5 presents the synthesis (*i.e.*, approximate decomposition) methods that can efficiently map arbitrary functions into the DALTA architecture. Section 6.6 shows the experimental results. Finally, Section 6.7 summarizes the chapter.

6.2 Preliminaries

This section presents the preliminaries related to this chapter. Specifically, traditional disjoint decomposition will be introduced as follows.

x_1x_2 \ x_3x_4	00	01	10	11	
00	0	1	1	0	$\phi \wedge \overline{x_1} \wedge \overline{x_2}$
01	1	0	0	1	$\bar{\phi} \wedge \overline{x_1} \wedge x_2$
10	1	1	1	1	$x_1 \wedge \overline{x_2}$
11	1	0	0	1	$\bar{\phi} \wedge x_1 \wedge x_2$

Figure 6.1: A 2-dimensional truth table, or Boolean matrix, of a function f.

Definition 6.1. Let f be a Boolean function of n variables and $X = \{x_1, \ldots, x_n\}$ be its inputs. Let $\{A, B\}$ be a partition on X. The function f has a *disjoint decomposition* with *free set* A and *bound set* B if there exist functions ϕ and F such that $f(X) = F(\phi(B), A)$. The function ϕ and F are called the *bound-set function* and the *free-set function*, respectively. If the function f has a disjoint decomposition, the function is said to be *decomposable*.

Not every Boolean function is decomposable. Ashenhurst gives a necessary and sufficient condition for the existence of a disjoint decomposition with a given partition on the input variables (Ashenhurst, 1959). It is based on a *2-dimensional (2-dimensional) truth table* representation of the Boolean function, in which some variables define the columns and the remaining variables define the rows; an example is shown in Fig. 6.1. In what follows, this representation is also called a *Boolean matrix*. The necessary and sufficient condition is given by the theorem below.

Theorem 6.1. *Let $\{A, B\}$ be a partition on X. A Boolean function f is decomposable with free set A and bound set B, if and only if the Boolean matrix with the variables in A and B defining the rows and the columns, respectively, has at most four distinct types of rows that can be classified into the following categories: 1) a pattern of all 0s; 2) a pattern of all 1s; 3) a fixed pattern p of 0's and 1's; 4) the complement of the pattern p.*

A proof of the above theorem can be found in the reference (Shen and Mckellar, 1970). The following example illustrates how to obtain the disjoint decomposition once the condition in Theorem 6.1 is satisfied.

Example 6.1. Fig. 6.1 shows a Boolean matrix of a Boolean function $f(x_1, x_2, x_3, x_4)$ with variables x_1, x_2 defining the rows and variables x_3, x_4 defining the columns. It satisfies the condition described in Theorem 6.1: row 1 falls into Category 3, rows 2 and 4 fall into Category 4, and row 3 falls into Category 2. Therefore, function f is decomposable with free set as $\{x_1, x_2\}$ and bound set as $\{x_3, x_4\}$. The truth table of the function $\phi(x_3, x_4)$ is set as the pattern in Category 3. For this example, the truth table is "0110" and correspondingly, $\phi(x_3, x_4) = \bar{x}_3 x_4 + x_3 \bar{x}_4$. The functions for the 1st, 2nd, 3rd, and 4th row of the Boolean matrix are shown on the right of the Boolean matrix in Fig. 6.1. The final expression of f is $f = \phi \bar{x}_1 \bar{x}_2 + \bar{\phi} \bar{x}_1 x_2 + x_1 \bar{x}_2 + \bar{\phi} x_1 x_2 = F(\phi, x_1, x_2)$.

6.3 Key Idea: Approximate Disjoint Decomposition

To facilitate later description, this section presents the key idea, which is *approximate disjoint decomposition*. As is shown later, if a function is decomposable, it can be implemented with a more efficient LUT structure.

However, an arbitrary function may not be decomposable. Nevertheless, if the outputs of some input patterns are deliberately changed, we can obtain a decomposable function. Such a technique is called approximate disjoint decomposition, or *approximate decomposition* in short.

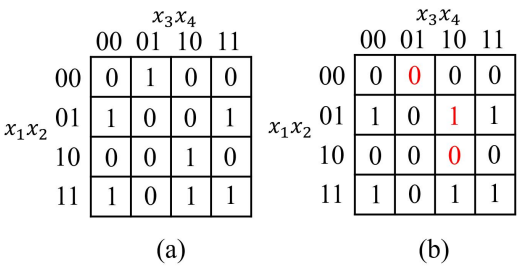


Figure 6.2: Boolean matrices of (a) a non-decomposable function and (b) a decomposable function that approximates the function in (a).

Example 6.2. Fig. 6.2(a) shows a single-output Boolean function represented as a Boolean matrix. From the matrix, it is shown that the function is not decomposable with the free set $\{x_1, x_2\}$ and the bound set $\{x_3, x_4\}$, since the condition in Theorem 6.1 is not satisfied. However, by flipping the outputs of some input patterns, a decomposable function can be constructed; an example is shown in Fig. 6.2(b) with the changed outputs shown in red.

As there are many ways to change the outputs, one that minimizes the error is preferred. In order to evaluate the error induced by a change, the occurrence probability of each input pattern should be known. This can be obtained by analyzing the typical input data set. Thus, in what follows, it is assumed that such probabilities are known.

It should be noted a previous work (Yao et al., 2017) also considers the approximate decomposition of a function. (Yao et al., 2017) proposed a technique to find approximate functions of maximum fanout-free cones (MFFCs), which is a special region in a circuit. However, the proposed method is quite different from theirs. First, their method is limited to single-output functions, while ours can deal with multiple-output functions. Second, their method is a heuristic one that finds the local optimum, while this chapter formulates the approximate decomposition problem into ILP and tries to find the global optimum. Finally, their method aims at designing logic circuits, while this chapter aims at finding patterns to be stored in the proposed approximate LUT architecture.

The following sections will first present DALTA, a LUT architecture supporting the computation based on decomposition. Then, it will be elaborated how to find the closest approximate decomposition for a given function so that it can be computed by DALTA.

6.4 Decomposition-Based Approximate LUT Architecture

This section presents DALTA, the proposed decomposition-based approximate LUT architecture.

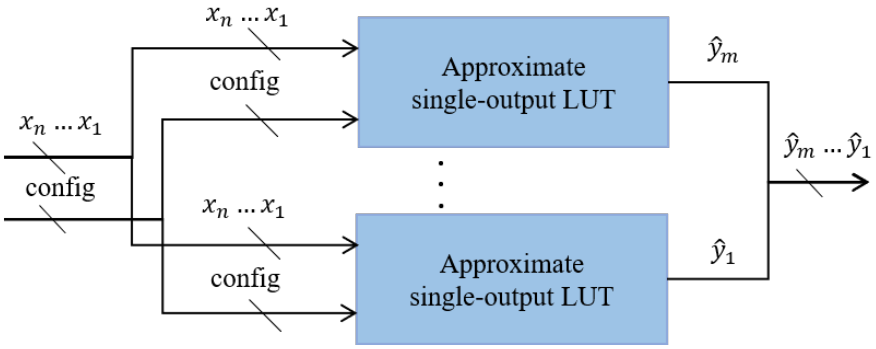


Figure 6.3: Overall decomposition-based approximate LUT architecture.

Let us consider the implementation of an n-input m-output Boolean function $Y = G(X)$, where $X = (x_1, \dots, x_n)$ with each $x_i \in \{0,1\}$, $Y = (y_1, \dots, y_m)$ with each $y_i \in \{0,1\}$, and $G = (g_1, \dots, g_m)$ with each g_i as an n-input single-output Boolean function. DALTA implements an approximation of G, denoted as $\hat{G} = (\hat{g}_1, \dots, \hat{g}_m)$, where $\hat{g}_i$ $(1 \le i \le m)$ is an n-input single-output Boolean function approximating g_i. In what follows, G and $\hat{G}$ are called *group functions* and g_i and $\hat{g}_i$ are called *component functions*.

DALTA is shown in Fig. 6.3. It consists of m identical *approximate single-output LUTs*, each implementing an n-input single-output Boolean function. These LUTs share the n-bit input X and the configuring signal *config* as inputs, and their outputs $\hat{y}_1, \dots, \hat{y}_m$ are combined into an m-bit output $\hat{Y}$.

Fig. 6.4 illustrates the structure of an approximate single-output LUT. It has three components: a LUT pair, a routing box, and a configuring box.

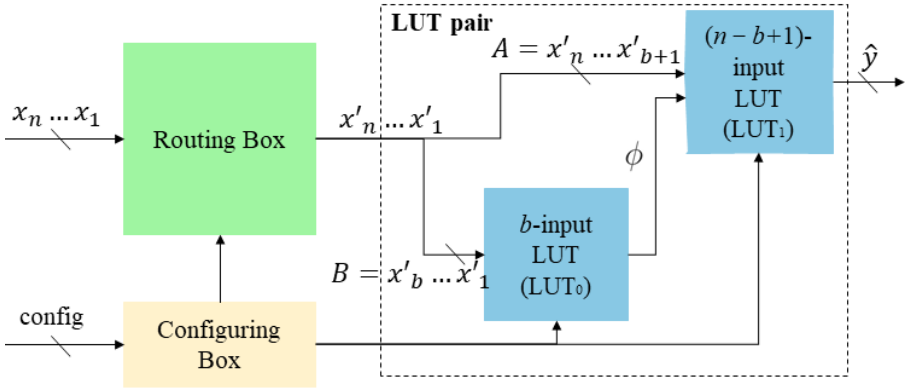


Figure 6.4: Approximate single-output LUT.

A LUT pair, as shown in Fig. 6.4, consists of a b-input LUT (LUT_0) and an $(n-b+1)$-input LUT (LUT_1).

A routing box shuffles the input $X = (x_1, \ldots, x_n)$ into $X' = (x'_1, \ldots, x'_n)$. The first b inputs of X', $x'_1, \ldots, x'_b$, are fed into LUT_0, while LUT_1 has its inputs as the remaining inputs $x'_{b+1}, \ldots, x'_n$ together with the output of LUT_0. The routing box can be easily implemented by n n-to-1 multiplexers (MUXs). The jth ($1 \leq j \leq n$) MUX is controlled by a $\lceil \log_2(n) \rceil$-bit selection signal to decide the routing from X to the jth output x'_j.

Based on the routing box and structure of the LUT pair, we can easily see that an approximate single-output LUT computes a single-output decomposable function

$$f(X) = F(\phi(B), A), \tag{6.1}$$

where A and B are the free set of size $(n - b)$ and the bound set of size b, respectively. If we can find an approximate decomposition of the function g_i, $\hat{g}_i$, satisfying the form shown in Eq. (6.1), we can implement $\hat{g}_i$ by the approximate single-output LUT.

Notably, implementing functions with the LUT pair reduces the storage cost dramatically. Storing the original function g_i requires a LUT of 2^n entries. In contrast, storing its approximation $\hat{g}_i$ using the LUT pair only needs $2^b + 2^{n-b+1}$ LUT entries. A typical configuration sets b as $\lceil \frac{n}{2} \rceil$ so that the costs of LUT_0 and LUT_1 are balanced. In this case, the LUT pair structure only requires $\frac{3}{2^{n/2}}$ and $\frac{1}{2^{(n-3)/2}}$ of entries compared to the accurate LUT structure for even and odd n, respectively.

The routing box provides design flexibility. It allows different approximate single-output LUTs to have different partitions of the free and bound sets. An alternative architecture is one with a single routing box shared by all the LUT pairs. This reduces the circuit area and power, but the configuration space is also reduced, which may cause an increase in approximation error.

Each approximate signal-output LUT is configured by a configuring box. It takes configuring signals *config* as input, including the free-set and bound-set functions to be stored in the LUT pair and the routing box configuration signals.

6.5 Efficient Synthesis Methods Based on Approximate Decomposition

As mentioned above, the proposed LUT architecture implements an approximation of the given group function G through the approximate decomposition of each component function g_i of G. This section elaborates efficient synthesis methods that can derive a close approximation for G. First, this section focuses on finding

approximate decomposition by considering each component function separately (see Section 6.5.1). Then, this section describes how to jointly decompose component functions by taking their significance into account (see Section 6.5.2).

6.5.1 Separate Decomposition of Component Functions

This section presents a method that finds the approximate decomposition for each component function g_k separately. First, a solution is proposed by assuming that g_k's free and bound sets are given. Later, it is discussed how to explore different partitions of the free and bound sets.

Let us focus on a particular component function g_k $(1 \leq k \leq m)$. The task is to find an approximate decomposition with the given variable partition that is closest to g_k. The proposed method is based on analyzing a Boolean matrix with variables in the free and bound sets defining the rows and the columns, respectively. The following notations will be used in the rest of this chapter.

- The variable partition of the component function g_k is denoted as $\Omega_k = \{A_k, B_k\}$, where A_k and B_k are the free set and the bound set, respectively. Note that $|A_k| = n - b$ and $|B_k| = b$.

- It is defined that $r = 2^{n-b}$ and $c = 2^b$. By the proposed architecture, r and c equal the numbers of rows and columns of the Boolean matrix, respectively.

- The uth $(1 \leq u \leq rc)$ input pattern of the component function g_k is denoted as X_{ku}. The outputs of g_k and $\hat{g}_k$ under X_{ku} are denoted as y_{ku} and $\hat{y}_{ku}$, respectively.

- The input pattern corresponding to the cell at row i $(1 \leq i \leq r)$ and column j $(1 \leq j \leq c)$ of the Boolean matrix under the variable partition of g_k is denoted as I_{kij}. I_{kij} occurs with probability p_{kij}. The outputs of g_k and $\hat{g}_k$ under I_{kij} are denoted as o_{kij} and $\hat{o}_{kij}$, respectively.

- The mapping from the index (i, j) of the 2-dimensional Boolean matrix under the variable partition of g_k to the index u of the 1D truth table is denoted as H. Specifically, $H(k, i, j) = u$, where k corresponds to the kth component function g_k. Then, we have

$$\begin{aligned} I_{kij} &= X_{k(H(k,i,j))}, \\ o_{kij} &= y_{k(H(k,i,j))}, \\ \hat{o}_{kij} &= \hat{y}_{k(H(k,i,j))}. \end{aligned} \tag{6.2}$$

A component function is a single-output function. A natural error measure for the approximation of a single-output function is ER, which is defined as the ratio of input patterns with a wrong output for the approximate function. Given the above notations, ER can be calculated as

$$ER = \sum_{i=1}^{r} \sum_{j=1}^{c} p_{kij} |o_{kij} - \hat{o}_{kij}|.$$

The task is to find a decomposable function $\hat{g}_k$ that minimizes the ER. By Theorem 6.1, when the free and bound sets are given, a decomposable function can be characterized by two variables of the Boolean matrix with the free and bound sets. The first is the row pattern in Category 3 of Theorem 6.1. It is represented as a vector $V = (v_1, \ldots, v_c) \in \{0,1\}^c$, which is called a *pattern vector*. The second is the collection of the category indices of all the rows. It is represented as a vector $T = (t_1, \ldots, t_r) \in \{1,2,3,4\}^r$, which is called a *row-type vector*. The entry t_i represents one of the four categories listed in Theorem 6.1 that the ith row belongs to. For example, for the decomposable function shown in Fig. 6.2(b), its pattern vector is $V = (1,0,1,1)$ and its row-type vector is $T = (1,3,1,3)$. Thus, in order to find the optimal decomposable function, we only need to find its corresponding pattern and row-type vectors.

Actually, for this problem, we only needed to solve the pattern vector. An optimization problem can be formulated as follows with the unknowns to be solved as v_j $(1 \le j \le c)$.

$$\min \textstyle\sum_{i=1}^{r} E_i, \tag{6.3}$$

$$s.t. \quad E_i = \min\{e_{i1}, e_{i2}, e_{i3}, e_{i4}\}, \text{ for } 1 \le i \le r, \tag{6.4}$$

$$e_{i1} = \textstyle\sum_{j=1}^{c} p_{kij} o_{kij}, \text{ for } 1 \le i \le r,$$

$$e_{i2} = \textstyle\sum_{j=1}^{c} p_{kij} (1 - o_{kij}), \text{ for } 1 \le i \le r,$$

$$e_{i3} = \textstyle\sum_{j=1}^{c} p_{kij} |v_j - o_{kij}|, \text{ for } 1 \le i \le r,$$

$$e_{i4} = \textstyle\sum_{j=1}^{c} p_{kij} |1 - v_j - o_{kij}|, \text{ for } 1 \le i \le r,$$

$$v_j \in \{0,1\}, \text{ for } 1 \le j \le c.$$

In the above formulation, e_{id} $(1 \le i \le r,\ 1 \le d \le 4)$ is the ER of row i of the Boolean matrix of $\hat{g}_k$ if it is in Category d. In order to minimize the ER of $\hat{g}_k$, row i of the Boolean matrix of $\hat{g}_k$ should belong to the category that minimizes the ER of that row. Correspondingly, the minimum ER for row i, E_i, is given by Eq. (6.4). The optimization target is the sum of E_i over all the rows (see Eq. (6.3)).

The above formulation can be further transformed into an ILP problem. The non-linear terms include

the absolute value function $|f|$ and the minimum function. The former can be transformed into a maximum function as $|f| = \max\{f, -f\}$. Then, the minimum and maximum functions can be further transformed into linear constraints (Kolman and Beck, 1995). Furthermore, the constraint $v_j \in \{0, 1\}$ can be relaxed to a linear constraint $0 \leq v_j \leq 1$ for further acceleration. Experiment results show that under the uniform input distribution, such relaxation does not affect optimality.

Up to now, we have obtained an approximate decomposition for a single-output component function g_k with a fixed bound and free set. Since different partitions of the bound and free sets can result in different errors between g_k and $\hat{g}_k$, different partitions should be explored. However, the total number of partitions grows exponentially with the input number. To limit the runtime, a parameter P is set as the maximum number of partitions to be explored. If the total number of partitions exceeds P, then P partitions will be randomly explored. Otherwise, all possible partitions are evaluated.

6.5.2 Joint Decomposition of Component Functions

The separate decomposition presented in Section 6.5.1 has a drawback that it ignores the different significance of the outputs $y_1, \ldots, y_m$. In many applications, the outputs encode a numerical value as $\sum_{k=1}^{m} 2^{k-1} y_k$. The ignorance of output significance can cause a large error. For example, consider an accurate output $(y_3, y_2, y_1) = (1, 0, 0)$ and its two approximations, $(0, 0, 0)$ and $(0, 1, 1)$. The separate decomposition prefers the first approximation over the second, as the last two bits of the first are still correct, while those of the second are both incorrect. However, in terms of the numerical value, the second should be preferred, as it is closer to the accurate numerical value than the first.

To address the issue of the separate decomposition, this section proposes a joint decomposition of all the component functions. The same notations in Section 6.5.1 are used here. It is also assumed that the outputs encode a numerical value in the binary radix format. The NMED is used to measure the error, which is calculated as follows:

$$NMED = \frac{1}{2^m - 1} \sum_{i=1}^{r} \sum_{j=1}^{c} p_{kij} \left| \sum_{k=1}^{m} 2^{k-1} \left(o_{kij} - \hat{o}_{kij} \right) \right| \tag{6.5}$$

The task is to find the joint approximate decomposition that minimizes the NMED. Next, two ILP-based solutions with different variable partition strategies and an efficient heuristic solution are presented.

Decomposition with Shared Variable Partition

The variable partition strategy used here is that all the component functions have the same partition. In this case, all the approximate single-output LUTs can share one routing box, thus, reducing the area and power.

As mentioned in Section 6.5.1, each approximate function $\hat{g}_k$ is fully determined by its pattern and row-type vectors. The pattern vector of $\hat{g}_k$ is denoted as $V_k = (v_{k1}, \ldots, v_{kc})$, and the row-type vector is denoted as $T_k = (t_{k1}, \ldots, t_{kr})$. To facilitate the ILP formulation, binary indicator variables s_{kid} ($1 \leq k \leq m$, $1 \leq i \leq r$, $1 \leq d \leq 4$) are introduced to encode a row-type vector. Specifically, $s_{kid} = 1$ if and only if $t_{ki} = d$, *i.e.*, the ith row in the Boolean matrix of $\hat{g}_k$ belongs to Category d. With the variables s_{kid}, the optimization problem can be formulated as follows.

$$\min \frac{1}{2^m - 1} \sum_{i=1}^{r} \sum_{j=1}^{c} p_{kij} \left| \sum_{k=1}^{m} 2^{k-1} (o_{kij} - \hat{o}_{kij}) \right| \tag{6.6}$$

$$s.t. \quad \hat{o}_{kij} = s_{ki0} \cdot 0 + s_{ki1} \cdot 1 + s_{ki2} v_{kj} + s_{ki3}(1 - v_{kj}), \tag{6.7}$$

$$\text{for } 1 \leq k \leq m, 1 \leq i \leq r, 1 \leq j \leq c,$$

$$\sum_{d=1}^{4} s_{kid} = 1, \text{ for } 1 \leq k \leq m, 1 \leq i \leq r, \tag{6.8}$$

$$s_{kid} \in \{0, 1\}, \text{ for } 1 \leq k \leq m, 1 \leq i \leq r, 1 \leq d \leq 4, \tag{6.9}$$

$$v_{kj} \in \{0, 1\}, \text{ for } 1 \leq k \leq m, 1 \leq j \leq c.$$

In the above formulation, Eq. (6.6) is same as Eq. (6.5), which calculates the NMED. Eq. (6.7) gives the output of the approximate function $\hat{g}_k$ under the input pattern I_{kij}, based on the definition of the binary indicator variables. Eqs. (6.8) and (6.9) are the basic constraints on the binary indicator variables s_{kid}. The above formulation includes two non-linear computations, an absolute value function, and the product of two binary variables. Fortunately, both of them can be transformed into linear constraints (Kolman and Beck, 1995). Thus, the above formulation can be transformed into an ILP problem. Similar to Section 6.5.1, at most P variable partitions are explored for runtime concern.

Decomposition with Non-shared Variable Partition

The variable partition strategy used here is that all the component functions can have different partitions. Compared with the strategy in Section 6.5.2, it explores more on the variable partitions and may achieve much smaller errors.

The method finds decomposition of each component function sequentially from the MSB to the LSB. When

decomposing the kth component function g_k, it is assumed that the rest component functions $\hat{g}_\lambda$ $(1 \leq \lambda \leq m, \lambda \neq k)$ have been known. In other words, the outputs of the rest component functions $\hat{y}_{\lambda u}$ $(1 \leq \lambda \leq m, \lambda \neq k, 1 \leq u \leq rc)$ have been already known. Then, under a given variable partition Ω_k, in order to find the optimal decomposable function $\hat{g}_k$, the optimization problem can be formulated as follows with the unknowns v_j $(1 \leq j \leq c)$.

$$\min \textstyle\sum_{i=1}^{r} E_i, \tag{6.10}$$

$$s.t. \quad E_i = \tfrac{1}{2^m - 1} \min\{e_{i1}, e_{i2}, e_{i3}, e_{i4}\}, \text{for } 1 \leq i \leq r, \tag{6.11}$$

$$u_{ij} = H(k, i, j), \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{6.12}$$

$$Y_{ij} = \textstyle\sum_{\lambda=1}^{m} 2^{\lambda-1} y_{\lambda u_{ij}}, \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{6.13}$$

$$\hat{Y}_{ij} = \textstyle\sum_{\lambda=1}^{k-1} 2^{\lambda-1} \hat{y}_{\lambda u_{ij}} + \sum_{\lambda=k+1}^{m} 2^{\lambda-1} \hat{y}_{\lambda u_{ij}}, \tag{6.14}$$

$$\text{for } 1 \leq i \leq r, 1 \leq j \leq c,$$

$$D_{ij} = \hat{Y}_{ij} - Y_{ij}, \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{6.15}$$

$$e_{i1} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} \right|, \text{for } 1 \leq i \leq r, \tag{6.16}$$

$$e_{i2} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1} \right|, \text{for } 1 \leq i \leq r, \tag{6.17}$$

$$e_{i3} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1} v_j \right|, \text{for } 1 \leq i \leq r, \tag{6.18}$$

$$e_{i4} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1} (1 - v_j) \right|, \text{for } 1 \leq i \leq r, \tag{6.19}$$

$$v_j \in \{0, 1\}, \text{for } 1 \leq j \leq c.$$

In the above formulation, Eq. (6.12) converts the index (i, j) of the 2-dimensional Boolean matrix to the index u_{ij} of the 1D truth table. Eq. (6.13) calculates the accurate output encoded by $g_1, \ldots, g_m$ under the input pattern I_{kij}. Eq. (6.14) computes the approximate output encoded by $\hat{g}_1, \ldots, \hat{g}_{k-1}, 0, \hat{g}_{k+1}, \ldots, \hat{g}_m$ under the input pattern I_{kij}. Eq. (6.15) gives the difference between the above two output values.

In Eqs. (6.16)–(6.19), e_{id} $(1 \leq i \leq r,\ 1 \leq d \leq 4)$ denotes the MED over the input patterns on row i of $\hat{g}_k$'s Boolean matrix if the row is in Category d. For example, consider e_{i3}. Since $d = 3$, the pattern vector $(v_1, \ldots, v_c)$ is selected as the output for the ith row of $\hat{g}_k$'s Boolean matrix. Thus, we have $\hat{o}_{kij} = v_j$ $(1 \leq j \leq c)$. By Eqs. (6.2) and (6.12), we further have $\hat{y}_{ku_{ij}} = v_j$. Combining this equation with Eqs. (6.13), (6.14), and (6.15), we have

$$D_{ij} + 2^{k-1} v_j = \sum_{\lambda=1}^{m} 2^{\lambda-1} (\hat{y}_{\lambda u_{ij}} - y_{\lambda u_{ij}}).$$

Combining the above equation with Eq. (6.16), we can conclude that e_{i3} indeed gives the MED over the input patterns on row i of $\hat{g}_k$'s Boolean matrix if the row is in Category 3.

In order to minimize the NMED, row i of the Boolean matrix should belong to the category that minimizes the MED of that row. Correspondingly, the minimum MED for row i, E_i, is given by Eq. (6.11). The optimization target is the sum of E_i over all the rows (see Eq. (6.10)). Similar to Section 6.5.1, the above formulation can be transformed into an ILP problem.

Since the error measure is NMED, the MSBs contribute more to the error. Therefore, the above optimization problem is iteratively solved from the MSB to the LSB, as shown in Algorithm 12. In other words, the approximate functions on the MSB, the second MSB, up to the LSB are solved sequentially. The inputs of Algorithm 12 include the iteration round R, the variable partition limit P, and the accurate component functions $Y = \{y_{ku} | 1 \leq k \leq m, 1 \leq u \leq rc\}$. The outputs include the best variable partitions for the component functions $\Omega^* = (\Omega_1, \ldots, \Omega_m)$, the corresponding pattern vectors $V^* = (V_1^*, \ldots, V_m^*)$, and the row-type vectors $T^* = (T_1^*, \ldots, T_m^*)$.

Note that the above optimization problem assumes that all the approximate component functions except for $\hat{g}_k$ are known, which is clearly not held initially. To solve this issue, the accurate component function is used as an estimate to the approximate component function (Line 1). Then, the algorithm iterates for R rounds, and for each round, it decomposes each component function from MSB to LSB. When finding the kth approximate component function $\hat{g}_k$, a set Φ of variable partition candidates of size no more than P (similar to Section 6.5.1) is generated (Line 5). Each partition candidate ω is evaluated by solving the optimization problem in Eq. (6.10) (Lines 6–8). The minimal NMED E_k^* under those partitions is kept, as well as the corresponding variable partition Ω_k^*, pattern vector V_k^*, and row-type vector T_k^* (Lines 9–10). After checking all the variable partitions in set Φ, the approximate function estimation $\hat{y}_{ku}$ is updated according to the kept Ω_k^*, V_k^*, T_k^* (Line 11). Finally, the best variable partitions, pattern vectors, and row-type vectors of all the component functions are put together and returned (Lines 12–13).

Heuristic Acceleration Technique

Unfortunately, the variable number of the previous ILP-based methods increases exponentially with the bound set size, as $c = 2^b$. For a target function with many inputs, it is challenging to find an optimal approximation for it by the ILP-based methods. This section proposes a heuristic method to search for a good local optimal solution.

Same as the decomposition method with non-shared variable partition, let us focus on finding an approximate

Algorithm 12: *Iterative_ Approximate_ Solution*(R, P, Y)

Input: Iteration round R, variable partition limit P,
accurate function $Y = \{y_{ku} | 1 \leq k \leq m, 1 \leq u \leq rc\}$.
Output: Best variable partitions Ω^*, best pattern vectors V^*, best row-type vectors T^*.

```
Approximate function estimation Ŷ = {ŷ_ku | 1 ≤ k ≤ m, 1 ≤ u ≤ rc} ← Y;
for round l from 1 to R do
    for k from m downto 1 do
        Best NMED at the kth bit E*_k ← ∞;
        Variable partitions Φ ← GenerVarPart(P);
        for each variable partition ω ∈ Φ do
            Pattern vector V_k ← SolveILP(Y, Ŷ, ω);
            Get corresponding row-type vector T_k and NMED E_k;
            if E_k < E*_k then
                (E*_k, Ω*_k, V*_k, T*_k) ← (E_k, ω, V_k, T_k);
        Ŷ_k = {ŷ_ku | 1 ≤ u ≤ rc} ← GetComponentFunction(Ω*_k, V*_k, T*_k);
(Ω*, V*, T*) ← ((Ω*_1, ..., Ω*_M), (V*_1, ..., V*_M), (T*_1, ..., T*_M));
return (Ω*, V*, T*)
```

component function $\hat{g}_k$ given that the rest component functions have been known. Thus, equivalently, we work on the Boolean matrix and try to find a pair of pattern and row-type vectors (V, T) giving the smallest NMED. The basic idea of the heuristic method is that if one of the pattern vectors V and the row-type vector T is fixed, the optimal choice for the other can be identified efficiently. If the pattern vector V is fixed, the row-type vector is updated element-wise. To decide t_i $(1 \leq i \leq r)$, the 4 categories are tried on the ith row, and set t_i as the category giving the smallest MED over the input patterns on row i of the Boolean matrix for $\hat{g}_k$, under the current V and current choice of the rest component functions. If the row-type vector T is fixed, the pattern vector is updated element-wise. To decide v_j $(1 \leq j \leq c)$, the value $v_j = 0$ and $v_j = 1$ are tried, and set v_j as the value that gives the smallest MED over the input patterns on the jth column of the Boolean matrix for $\hat{g}_k$, under the current T and current choice of the rest component functions.

Thus, instead of searching for V and T simultaneously, the proposed method will update V and T alternatively by fixing one and optimizing the other until they converge to fixed values. Then, a local optimal solution is obtained. Since the final solution highly depends on the initial choice for V, the proposed method tries different initial V's for Z times and finally returns the best local optimum.

After elaborating all four approximate decomposition methods, their complexity is compared, as is shown in Table 6.1. The joint decomposition method with shared variable partition has the most variables and constraints, so it suffers from a long runtime. The separate decomposition method, the joint decomposition method with non-shared variable partition, and the heuristic joint decomposition method have the same c unknown variables, and hence, they are more efficient.

Table 6.1: Comparison of different approximate decomposition methods.

Method	Separate	Joint (shared)	Joint (non-shared)	Joint (heuristic)
Objective	Eq. (6.3)	Eq. (6.6)	Eq. (6.10)	Eq. (6.10)
#variables	c	$m(c+4r)$	c	c
#constraints	$c+5r$	$m(c+5r+rc)$	$c+5r+4rc$	$-$

6.6 Experimental Results

This section presents the experimental results of DALTA.

6.6.1 Experiment Setup

All the LUT architectures are implemented in Verilog Hardware Description Language, where LUTs are implemented by RAMs consisting of D flip-flops. Synopsys Design Compiler (Synopsys, Inc., 2022) is used for synthesizing the architectures, mapping them into Nangate 45nm standard cell library (Nangate, Inc., 2022), and measuring their areas and delays. The function of the architectures is verified by Synopsys VCS (Synopsys, Inc., 2022), and the power is evaluated by Synopsys PrimeTime (Synopsys, Inc., 2022). In addition, the approximate decomposition methods are implemented in C++. Gurobi (Gurobi, 2021) is used as the ILP solver. All the experiments are performed on a computer with an 8-core 3.8GHz AMD 3700X processor and 16GB RAM.

DALTA is compared with two other architectures. The first is a rounding-based approximate LUT architecture. A q-bit rounding-based LUT architecture just rounds the q LSBs of the outputs and keeps the $(m-q)$ MSBs. The second is a state-of-the-art architecture, ApproxLUT (Tian et al., 2017). For ApproxLUT, the pattern selection algorithm proposed in the ApproxLUT paper is re-implemented to decide the stored patterns in it. Its hardware part, consisting of a LUT, a pattern matching unit, and arithmetic units, is reproduced by Verilog.

All continuous and non-continuous functions listed in Table 6.2 are tested, where the continuous functions are from the reference (Tian et al., 2017) and the non-continuous ones include 4 arithmetic circuits from AxBench (Yazdanbakhsh et al., 2016). To compute continuous functions with n-input and m-output LUTs, their inputs are quantized into n bits and outputs into m bits. Since some circuits in AxBench have too many inputs that are not suitable for LUT-based computing, their inputs are re-quantized into 16 bits and adjust the number of outputs correspondingly. Actually, the four functions from AxBench take two 8-bit operands x_1 and x_2 as inputs, and they are stitched together to form a 16-bit input x.

In this way, the output function $f(x)$ is a piecewise function with many segments, and hence it is non-continuous. For all the experiments, it is assumed that the inputs are uniformly distributed.

Important parameters of the proposed method are listed below. The variable partition limit is $P = 1000$. The iteration round in Algorithm 12 is $R = 5$. The number of different initial pattern vectors for the heuristic joint decomposition method is $Z = 30$. For the ILP solver Gurobi, the runtime limit to solve a single ILP problem is 3600s. If the limit is reached, Gurobi returns the current best solution with the smallest error.

Table 6.2: Benchmarks used in the experiments.

Continuous function	Domain	Range	Application
$\cos(x)$	$[0, \frac{\pi}{2}]$	$[0, 1]$	Geometry
$\tan(x)$	$[0, \frac{2\pi}{5}]$	$[0, 3.08]$	Geometry
$\exp(x)$	$[0, 3]$	$[0, 20.09]$	Financial
$\ln(x)$	$[1, 10]$	$[0, 1.61]$	Financial
$\text{erf}(x)$	$[0, 3]$	$[0, 1]$	Statistics
$\text{denoise}(x)$	$[0, 3]$	$[0, 0.81]$	Medical
Non-continuous function	**#input**	**#output**	**Application**
Brent-Kung	16	9	Arithmetic
Forwardk2j	16	16	Robotics
Inversek2j	16	16	Robotics
Multiplier	16	16	Arithmetic

6.6.2 Comparison of Different Decomposition Methods

Section 6.5 proposes four approximate decomposition methods: the separate decomposition method, the joint shared-partition decomposition method, the joint non-shared-partition decomposition method, and the joint heuristic method. Their performance is compared on the 6 continuous functions listed in Table 6.2. This experiment quantizes the functions into $n = 9$ inputs and $m = 9$ outputs. The free set size is 4 and the bound set size is $b = 5$, so the LUT sizes of the free-set and the bound-set functions are both 32. For the shared-partition method, there are so many variables (see Table 6.1) that exploration of different variable partitions for it is impractical. Therefore, the shared-partition method is applied only once with the partition found by a heuristic approach.*

For the other three methods, different variable partitions for them are explored, and the number of partitions is controlled by the parameter $P = 1000$.

Table 6.3 compares the performance of different decomposition methods. All numbers are the averages over the 6 benchmarks in Table 6.2. The last column reports the runtime of solving a single ILP problem under a fixed variable partition. There are three observations from the table. First, the shared-partition method

*Specifically, it is a modified version of the heuristic method. The modification is letting the approximate component functions $\hat{g}_k$'s share the same row-type vector T. The variable partition returned by the modified method is used here.

Table 6.3: Performance of various decomposition methods.

Method	NMED	Area	Route box area	Latency	Power	Runtime
Separate	1.93%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	2.21s
Shared	**0.49%**	**4823**μm^2	**335**μm^2	0.16ns	**0.25mW**	3600s
Non-shared	0.68%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	4.38s
Heuristic	0.70%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	**0.6ms**

produces approximate functions with the smallest NMED. The reason is that the shared-partition method directly minimizes NMED by taking all the outputs into account, which can avoid inducing errors on the most significant output bits. Second, the shared-partition method requires a smaller area and power than the others, and all the methods have the same latency. It is reasonable since the shared-partition method allows sharing a routing box over different LUT pairs in DALTA. However, the other methods need different partitions on the 9 outputs, and 9 copies of the routing boxes are needed. Third, the heuristic method is the most efficient and does not induce a large error, so it is preferred especially for more complex functions with larger input and output numbers.

6.6.3 Experiments on Continuous Functions

The benchmarks used in this experiment are the 6 continuous functions in Table 6.2. Different from Section 6.6.2, the inputs and outputs are quantized into $n = m = 16$ bits. In this case, only the heuristic joint decomposition method can obtain solutions in a reasonable time (*i.e.*, within a few hours), and it is chosen to generate approximate decompositions. The free set size is 7 and the bound set size is $b = 9$. Thus, the LUT sizes of the free-set and the bound-set functions are 256 and 512, respectively.

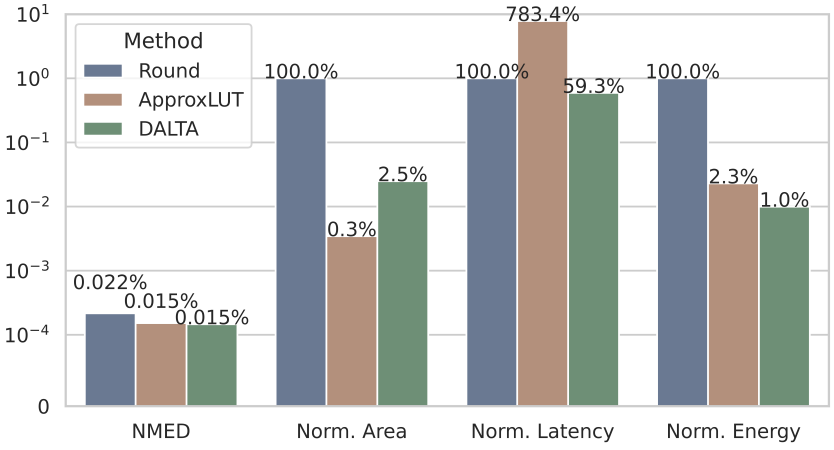


Figure 6.5: Comparison of different approximate LUT architectures on continuous functions.

DALTA is compared with ApproxLUT and the rounding-based LUT architecture. For ApproxLUT, the number of stored patterns is tuned to make its NMED the same as that of DALTA. Similarly, for rounding-based LUT architecture, the number of rounding bits is adjusted so that its NMED is slightly larger than that of DALTA. Fig. 6.5 compares the average NMEDs, areas, latencies, and energies over the 6 functions for the rounding architecture, ApproxLUT, and DALTA. The latency and the energy are measured for a single reading operation. For area, latency, and energy, the values are normalized to that of the rounding architecture. As shown in Fig. 6.5, the average NMEDs of the rounding architecture, ApproxLUT, and DALTA are $0.022\%, 0.015\%$, and 0.015%, respectively. In other words, the errors mainly distribute on the 4 LSBs. Compared to the rounding architecture, DALTA reduces area, latency, and energy by 97.5%, 40.7%, and 99%, respectively. Compared to ApproxLUT, DALTA improves latency and energy by 92.4% and 56.5%, respectively, but requires $8\times$ area.

6.6.4 Experiments on Non-Continuous Functions

The benchmarks used in this experiment are the 4 non-continuous functions in Table 6.2. They all have $n = 16$ inputs, but their output numbers m are different. The free set size is 7 and the bound set size is $b = 9$. The heuristic joint decomposition method is used to decompose these functions.

First, it is shown that ApproxLUT is not suitable to approximate non-continuous functions. The number of stored patterns in ApproxLUT is tuned so that the LUT sizes of ApproxLUT and DALTA are the same. Notably, DALTA consumes less energy than ApproxLUT when they have the same LUT size since ApproxLUT requires additional pattern matching and arithmetic units. The LUT size of DALTA is $(2^b + 2^{n-b+1})m = 768m$, while that of ApproxLUT is $3sm$, where s is the number of stored patterns. The parameter s is set as 256, so the LUT size of both architectures is the same. Table 6.4 compares the NMEDs between DALTA and ApproxLUT with the same LUT size for each non-continuous function. It is shown that DALTA can approximate non-continuous functions with smaller NMEDs, while ApproxLUT suffers from extremely large errors. The reason is that ApproxLUT utilizes first-order Taylor expansion to approximate functions, and it does not work on non-continuous functions.

Now, DALTA on non-continuous functions is compared with the rounding architecture. Similar to Section 6.6.3, the number of rounding bits is adjusted so that the rounding architecture has a slightly larger NMED than DALTA. Fig. 6.6 compares the average NMEDs, areas, latencies, and energies over the 4 functions for the two architectures. Again, the latency and the energy is measured for a single reading operation. For area, latency, and energy, the values are normalized to that of the rounding architecture. As shown in Fig. 6.6,

Table 6.4: Comparison of NMEDs of DALTA and ApproxLUT with the same LUT size.

Benchmark	NMED		LUT size in bits	
	DALTA	ApproxLUT	DALTA	ApproxLUT
Brent-Kung	**0.0978%**	84.8722%	6912	6912
Forwardk2j	**0.8955%**	61.3607%	12288	12288
Inversek2j	**0.6068%**	13.4480%	12288	12288
Multiplier	**0.6548%**	41.9141%	12288	12288

the average NMEDs over the 4 functions of the rounding architecture and DALTA are 0.81% and 0.56%, respectively. Compared to the rounding architecture, DALTA improves the area, latency, and energy by 95.8%, 39.0%, and 98.3%, respectively.

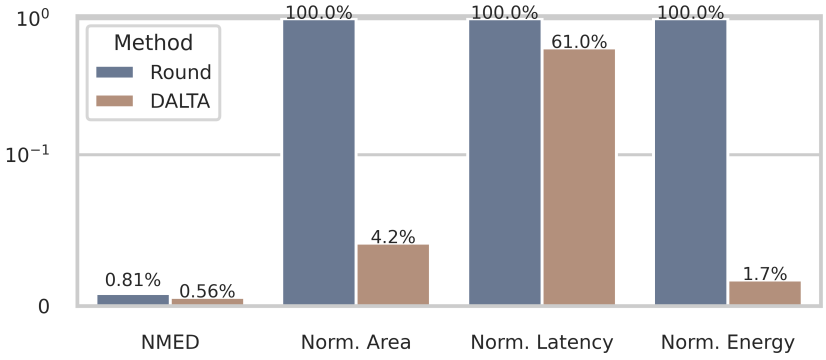


Figure 6.6: Comparison of DALTA and the rounding architecture on non-continuous functions.

6.6.5 Exploration of Different Structures of DALTA

Table 6.5: Performance of heuristic method for different configurations of DALTA for the $\cos(x)$ function.

#In/#Out	NMED	Area impr.	Latency impr.	Power impr.	Runtime/s
8/16	0.230%	3.0×	1.7×	4.5×	3
9/16	0.121%	4.3×	1.9×	6.4×	6
10/16	0.124%	6.2×	1.8×	9.2×	25
11/16	0.052%	9.4×	2.2×	13.9×	225
12/16	0.060%	13.6×	2.4×	20.2×	546
13/16	0.028%	20.7×	2.6×	30.8×	1036
14/16	0.027%	29.2×	2.5×	43.3×	2212
15/16	0.013%	45.0×	2.7×	66.7×	3622
16/16	0.014%	62.4×	1.6×	92.5×	6367

To show the exponential decrease in area and power with the input numbers, the heuristic method is applied to generate approximations for the $\cos(x)$ function targeting at different configurations of DALTA with different

input and output numbers. For an n-input function, the free set size is $\lfloor \frac{n}{2} \rfloor$, and the bound set size is $\lceil \frac{n}{2} \rceil$. Table 6.5 shows the performance of the heuristic method for different DALTA configurations. The cost (area, power, and delay) improvement is calculated as the cost of accurate LUT over that of DALTA. As the input and output numbers increase, the NMED roughly decreases, and the area improvement, power improvement, and runtime increase exponentially. It is not surprising because the proposed method enjoys an exponential decrease in LUT size over the accurate LUT. The reason for the long runtime is that as the input number increases, the numbers of rows and columns in Boolean matrices grow exponentially. From Section 13, the row-type and pattern vectors are both updated element-wise. Thus, the runtime shows exponential growth.

6.7 Summary

This chapter proposes DALTA, a decomposition-based approximate LUT architecture. The main idea is to decompose a function approximately into bound-set and free-set functions, and store them with LUTs of smaller sizes. Efficient synthesis algorithms are also developed to map an arbitrary function into a high-quality implementation on DALTA. The synthesis problem is formulated as an optimization problem, which is solved by ILP-based and heuristic-based methods. The experimental results show that DALTA significantly improves the area, delay, and power by introducing a small error.


20006301

Chapter 7

Conclusions and Future Works

In conclusion, this dissertation presents efficient synthesis methods for high-quality designs at the circuit and architecture layers.

At the circuit layer, this dissertation proposes efficient ALS methods to improve the quality of approximate circuits. For one thing, this dissertation studies how to improve the quality of the circuits generated by ALS. Chapter 2 introduces an area-driven ALS flow called ALSRAC. It devises an effective LAC based on approximate resubstitution to simplify the structures of approximate circuits, hence reducing the circuit area. Chapter 3 presents advanced ordering search algorithms for ALS. It employs beam search and MCTS to discover better orders of applying LACs, leading to smaller circuit areas. Chapter 4 proposes a delay-driven ALS flow called HEDALS. It iteratively shortens critical paths by applying optimized sets of LACs on the paths, thereby reducing the circuit delay. For another, this dissertation investigates how to enhance the efficiency of ALS. Given that error evaluation is the most time-consuming step in ALS, Chapter 5 introduces an efficient maximum error checking technique called MECALS to accelerate the step. An efficient ALS flow based on MECALS is developed, significantly reducing the runtime of ALS.

At the architecture layer, this dissertation designs high-quality approximate computing architecture and its supporting synthesis algorithms. Specifically, Chapter 6 proposes a low-power and high-speed approximate LUT architecture called DALTA. To implement an arbitrary function using the DALTA architecture, efficient synthesis methods are developed, aiming to find an optimal implementation with the smallest error.

7.1 Summary of Contributions

The major contributions of this dissertation are summarized as follows.

- At the circuit layer, this dissertation develops efficient ALS methods to generate high-quality approximate circuits with small areas and delays.

  - Chapter 2 proposes ALSRAC, an effective area-driven ALS framework. It employs a novel LAC based on approximate resubstitution, which has strong expressive power of approximate functions and provides a fine-grained way of approximation. An efficient simulation-based approach is devised to generate such LACs. The experimental results show that compared to state-of-the-art methods, ALSRAC can save more area by 7%∼18%.

  - Chapter 3 proposes two advanced ordering search techniques for ALS to reduce the circuit area. These techniques are based on beam seach and MCTS, both of which explore more possibilities for circuit simplification. The experimental results show that compared to the basic greedy search, the proposed techniques are effective in finding better LAC orderings and can achieve larger area reduction.

  - Chapter 4 proposes HEDALS, a highly efficient delay-driven ALS framework. HEDALS incorporates CEG to facilitate finding an optimized set of LACs , which can reduce the circuit delay while inducing a small error. Utilizing CEG, a maximum flow-based method and a priority cut-based method are developed to find optimized LAC sets. The experimental results show that compared to the state-of-the-art method, HEDALS can reduce the circuit delay by 32.3% and is 167× faster.

  - Chapter 5 proposes MECALS, an efficient maximum error checking method to accelerate ALS. MECALS utilizes PBD to build a theoretical foundation for maximum error checking and then employs SAT sweeping to efficiently check the maximum errors of all LACs. Based on MECALS, an efficient ALS flow is developed. The experimental results show that compared to the state-of-the-art ALS flow, the proposed flow is 13× faster and saves more area and delay.

- At the architecture layer, Chapter 6 proposes DALTA, a high-quality approximate LUT architecture based on approximate decomposition. Unlike previous approximate LUT architectures, DALTA can support arbitrary functions. To implement functions with DALTA, efficient synthesis methods based on ILP and heuristic techniques are devised. The experimental results show that DALTA achieves 56.5% energy savings and 92.4% latency reduction for continuous functions, as well as 39.0% energy savings and 98.3% latency reduction for non-continuous functions.

7.2 Future works

In the future, the author will continue his academic career and try to solve the following problems to make approximate computing more practical.

- Automatic synthesis of accuracy-configurable approximate circuits. Compared to an approximate circuit with a fixed error, a reconfigurable design is not fixed at design time. Its accuracy can be tuned according to external (*e.g.*, battery level) and internal (*e.g.*, output confidence) factors (Scarabottolo et al., 2020). Thus, a reconfigurable design is more flexible and practical. Although accuracy-configurable adders and multipliers have been proposed, research in the automatic synthesis of accuracy-configurable circuits is still preliminary, such as the method proposed in the reference (Venkataramani et al., 2013). In the future, the author will study ALS and AHLS methods that can synthesize accuracy-configurable implementations for arbitrary circuits.

- Formal verification of approximate designs. Verifying the error of large approximate designs with numerous inputs is very difficult. As we know, the error metrics can be classified into average and maximum errors. Although the formal verification of the maximum error can be converted into a SAT problem, the formal verification of the average error is still hard to be solved. Researchers proposed BDD-based methods to compute the average errors of approximate designs exactly, but they can only support small circuits such as 32-bit adders and 8-bit multipliers. How to efficiently evaluate average errors of large approximate designs is still challenging. In the future, the author will try to solve it.

- Cross-layer design. As discussed before, approximate computing techniques can be applied to the circuit, architecture, and software layers. However, each technique usually focuses on one of the layers. Combining the approximate techniques and performing cross-layer designing is very challenging. For example, we can consider both the circuit and architecture layers when designing an approximate processor. For one thing, we can apply aggressive voltage scaling at the architecture layer. For another, we can introduce function approximation at the circuit layer. However, both voltage scaling and function approximation will introduce errors. It is difficult to build a proper model to analyze the combined error induced by the cross-layer approximation. In the future, the author will study cross-layer designs to further reduce the hardware cost of computing systems.


20006301

Acknowledgements

After an unforgettable journey in the past five years, I am proud to say that the end of my Ph.D. period is finally in sight. Along the way, I have gained an abundance of knowledge and academic skills that have prepared me for the challenges ahead. However, it is the lifelong lessons and invaluable memories that have enriched my soul and made this journey more fulfilling. As I take a moment to look back, I find myself overwhelmed with gratitude, and I am willing to express my appreciation to all those who made this journey possible.

I would like to start by expressing my heartfelt gratitude to my advisor, Prof. Weikang Qian. He has been my guiding light, offering unwavering support, guidance, and encouragement throughout my doctoral journey. I am forever grateful for his belief in my abilities and for providing me with the tools, skills, and mindset to succeed. His attention to detail, enthusiasm for research, and constructive feedback have been invaluable in developing and refining my work. I am honored to have had Prof. Qian as my advisor and appreciate his consistent support.

Next, I would like to express my profound appreciation to my thesis committee members - Prof. Zhigang Ji, Prof. Honglan Jiang, Prof. Paul Weng, and Prof. An Zou. Their expertise and experience in the field have significantly contributed to the refinement of my research, making it more comprehensive. I am grateful for their guidance and support, especially during the dissertation writing and defense scheduling processes. I could not have achieved the level of quality in my dissertation without their assistance.

I also want to thank the members of the emerging computing technology lab for their support and help. They are Yi Wu, Chen Wang, Sanbao Su, Meng Yang, Kuncai Zhong, Xuan Wang, Shanshan Han, Weihua Xiao, Ziqi Meng, Xingyue Qian, Jian Shi, Xianjue Cai, Ruicheng Dai, Zexi Li, and Wenhui Liang. Their diverse perspectives and feedback have shaped my work, and I appreciate their support.

I also want to express my gratitude to other people who helped me with my research. Specifically, I would like to thank Prof. Paul Weng from UM-SJTU Joint Institute and Dr. Alan Mishchenko from UC Berkeley for providing me with important ideas and guidance. Additionally, I want to thank the undergraduates from

UM-SJTU Joint Institute who participated in my research. Their willingness to share their ideas with me is highly appreciated. Special thanks to Zhuangzhuang Zhou, Hanyu Wang, Zhiyuan Xiang, Niyiqiu Li, Jiajun Sun, Yuqi Mai, Yifan Qian, Sijun Tao, Haoxiang Jin, Xiaomi Zhou, Xiyu Liang, Jingjing Zhu, Haojia Sun, Xiaotian Shi, Hangxin Chen, and many others.

I also want to show my appreciation to my family for their constant love and support during my doctoral journey. They have been my pillars of strength, inspiring me to push beyond boundaries and work towards achieving my academic goals. Their unwavering support means the world to me. I am grateful for their patience, encouragement, and understanding throughout this journey.

Last but not least, I want to express my sincere gratitude to my girlfriend, Yuhang Chen, for her love, understanding, and support. She is always a source of strength for me in the most challenging times, providing me with support and motivation. Her feedback and critical input on my drafts played a vital role in shaping my work. I am thankful for the time she took out of her busy schedule to help me.

In conclusion, I would like to express my deepest gratitude to everyone who helps me achieve this milestone. I am honored to have had the opportunity to work with such wonderful people. I hope to continue to build on the knowledge and skills that I have gained during my doctoral journey and make meaningful contributions to the field of EDA in the future.

Publication List

Journal papers

[1]. **Chang Meng**, Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Yuhang Chen, Weikang Qian, "HEDALS: highly efficient delay-driven approximate logic synthesis," to appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2023.

[2]. Sanbao Su*, **Chang Meng***, Fan Yang, Xiaolong Shen, Leibin Ni, Wei Wu, Zhihang Wu, Junfeng Zhao, and Weikang Qian, "VECBEE: a versatile efficiency-accuracy configurable batch error estimation method for greedy approximate logic synthesis," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 41, no. 11, pp. 5085-5099, Nov. 2022. (*These authors contributed to the work equally and should be regarded as **co-first authors**.)

Conference Papers

[1]. **Chang Meng**, Weikang Qian, and Alan Mishchenko, "ALSRAC: approximate logic synthesis by resubstitution with approximate care set," in IEEE/ACM Design Automation Conference (DAC), 2020, pp. 1-6. (Acceptance rate: 23.2%.)

[2]. **Chang Meng***, Xuan Wang*, Jiajun Sun, Sijun Tao, Wei Wu, Zhihang Wu, Leibin Ni, Xiaolong Shen, Junfeng Zhao, and Weikang Qian, "SEALS: sensitivity-driven efficient approximate logic synthesis," in IEEE/ACM Design Automation Conference (DAC), 2022, pp. 1-6. (*These authors contributed to the work equally and should be regarded as **co-first authors**. Acceptance rate: 23%.)

[3]. **Chang Meng**, Zhiyuan Xiang, Niyiqiu Liu, Yixuan Hu, Jiahao Song, Runsheng Wang, Ru Huang, and Weikang Qian, "DALTA: a decomposition-based approximate lookup table architecture," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2021, pp. 1-8. (Acceptance rate: 23.5%.)

[4]. **Chang Meng**, Jiajun Sun, Yuqi Mai, Weikang Qian, "MECALS: a maximum error checking technique for approximate logic synthesis", in IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023, pp. 1-6. (Acceptance rate: 25%.)

[5]. **Chang Meng**, Paul Weng, Sanbao Su, and Weikang Qian, "Advanced ordering search for multi-level approximate logic synthesis," in IEEE/ACM International Workshop on Logic and Synthesis (IWLS), 2019, pp. 89-96.

[6]. Xingyue Qian, **Chang Meng**, Xiaolong Shen, Junfeng Zhao, Leibin Ni, and Weikang Qian, "High-accuracy low-power reconfigurable architectures for decomposition-based approximate lookup table", in IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023, pp. 1-6.

[7]. Yifan Qian, **Chang Meng**, Yawen Zhang, Weikang Qian, Runsheng Wang, and Ru Huang, "Approximate logic synthesis in the loop for designing low-power neural network accelerator," in IEEE/ACM International Symposium on Circuits and Systems (ISCAS), 2021, pp. 1-5.

[8]. Zuodong Zhang, Runsheng Wang, Zhe Zhang, Ru Huang, **Chang Meng**, Weikang Qian, and Zhuangzhuang Zhou, "Reliability-enhanced circuit design flow based on approximate logic synthesis," in IEEE/ACM Great Lakes Symposium on VLSI (GLVLSI), 2020, pp. 1-6.

[9]. Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Sanbao Su, **Chang Meng**, and Weikang Qian, "DALS: delay-driven approximate logic synthesis," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1-7.

Bibliography

Ansari, M. S., B. F. Cockburn, and J. Han (2019). A hardware-efficient logarithmic multiplier with improved accuracy. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 928–931.

Ashenhurst, R. L. (1959). The decompositions of switching functions. In *IEEE International Symposium on the Theory of Switching Functions*, pp. 74–116.

Asmuth, J. and M. L. Littman (2012). Learning is planning: near Bayes-optimal reinforcement learning via Monte-Carlo tree search. *arXiv preprint arXiv:1202.3699 1*, 1–8.

Balla, R.-K. and A. Fern (2009). UCT for tactical assault planning in real-time strategy games. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 40–45.

Bjarnason, R., A. Fern, and P. Tadepalli (2009). Lower bounding klondike solitaire with monte-carlo planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, Volume 19, pp. 26–33.

Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games (T-CIAIG) 4*(1), 1–43.

Camus, V., M. Cacciotti, J. Schlachter, and C. Enz (2018). Design of approximate circuits by fabrication of false timing paths: The carry cut-back adder. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS) 8*(4), 746–757.

Češka, M., J. Matyáš, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar (2017). Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 416–423.

Chandrasekharan, A., M. Soeken, D. Große, and R. Drechsler (2016). Approximation-aware rewriting of AIGs for error tolerant applications. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8.

Chatterjee, S., A. Mishchenko, and R. Brayton (2006). Factor cuts. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 143–150.

Chiang, A. C., I. S. Reed, and A. V. Banes (1972). Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Transactions on Computers (TC) 100*(2), 189–195.

Cho, K., Y. Lee, Y. H. Oh, G.-c. Hwang, and J. W. Lee (2014). eDRAM-based tiered-reliability memory with applications to low-power frame buffers. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 333–338.

Cong, J., C. Wu, and Y. Ding (1999). Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *ACM International Symposium on Field Programmable Gate Arrays (ISFPGA)*, pp. 29–35.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to algorithms*. MIT press.

Curtis, H. A. (1962). *A New Approach to the Design of Switching Circuits*. Van Nostrand.

De Dinechin, F. and A. Tisserand (2005). Multipartite table methods. *IEEE Transactions on Computers (TC) 54*(3), 319–330.

EPFL LSI Lab (2021). The EPFL combinational benchmark suite. https://lsi.epfl.ch/page-102566-en-html/benchmarks/. Accessed September 1, 2021.

Esmaeilzadeh, H., A. Sampson, L. Ceze, and D. Burger (2012). Architecture support for disciplined approximate programming. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 301–312.

Esposito, D., A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra (2018). Approximate multipliers based on new approximate compressors. *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I) 65*(12), 4169–4182.

Fang, Y., H. Li, and X. Li (2012). SoftPCM: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write. In *IEEE Asian Test Symposium (ATS)*, pp. 131–136. IEEE.

Froehlich, S., D. Große, and R. Drechsler (2017). Error bounded exact BDD minimization in approximate computing. In *IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 254–259.

Gupta, V., D. Mohapatra, A. Raghunathan, and K. Roy (2012). Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 32*(1), 124–137.

Gurobi (2021). Gurobi - the fastest solver. https://www.gurobi.com/. Accessed May 1, 2021.

Han, J. and M. Orshansky (2013). Approximate computing: An emerging paradigm for energy-efficient design. In *IEEE European Test Symposium (ETS)*, pp. 1–6.

Hansen, M. C., H. Yalcin, and J. P. Hayes (1999). Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers 16*(3), 72–80.

Hashemi, S., H. Tann, and S. Reda (2018). BLASYS: Approximate logic synthesis using Boolean matrix factorization. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6.

Hsiao, S.-F., C.-S. Wen, Y.-H. Chen, and K.-C. Huang (2016). Hierarchical multipartite function evaluation. *IEEE Transactions on Computers (TC) 66*(1), 89–99.

Hu, J. and W. Qian (2015). A new approximate adder with low relative error and correct sign calculation. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1449–1454.

Imani, M., D. Peroni, and T. Rosing (2017). CFPU: Configurable floating point multiplier for energy-efficient computing. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6.

Imani, M., A. Rahimi, and T. S. Rosing (2016). Resistive configurable associative memory for approximate computing. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1327–1332.

Jiang, H., J. Han, F. Qiao, and F. Lombardi (2015). Approximate radix-8 booth multipliers for low-power and high-performance operation. *IEEE Transactions on Computers (TC) 65*(8), 2638–2644.

Jiang, H., C. Liu, L. Liu, F. Lombardi, and J. Han (2017). A review, classification, and comparative evaluation of approximate arithmetic circuits. *ACM Journal on Emerging Technologies in Computing Systems (JETC) 13*(4), 1–34.

Kahng, A. B. and S. Kang (2012). Accuracy-configurable adder for approximate arithmetic designs. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 820–825.

Kocsis, L. and C. Szepesvári (2006). Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pp. 282–293.

Kocsis, L., C. Szepesvári, and J. Willemson (2006). Improved Monte-Carlo search. *University of Tartu, Estonia, Technical Report 1*, 1–22.

Kolman, B. and R. E. Beck (1995). *Elementary Linear Programming with Applications*. Academic Press.

Kulkarni, P., P. Gupta, and M. Ercegovac (2011). Trading accuracy for power with an underdesigned multiplier architecture. In *IEEE International Conference on VLSI Design (VLSID)*, pp. 346–351.

Kyaw, K. Y., W. L. Goh, and K. S. Yeo (2010). Low-power high-speed multiplier for error-tolerant application. In *IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, pp. 1–4.

Lai, Y.-T., M. Pedram, and S. B. Vrudhula (1993). BDD based decomposition of logic functions with application to FPGA synthesis. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 642–647.

Lee, R.-R., J.-H. R. Jiang, and W.-L. Hung (2008). Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 636–641.

Lee, S. and A. Gerstlauer (2017). Data-dependent loop approximations for performance-quality driven high-level synthesis. *IEEE Embedded Systems Letters (ESL) 10*(1), 18–21.

Li, C., W. Luo, S. S. Sapatnekar, and J. Hu (2015). Joint precision optimization and high level synthesis for approximate computing. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6.

Lin, C.-H. and C. Lin (2013). High accuracy approximate multiplier with error correction. In *IEEE International Conference on Computer Design (ICCD)*, pp. 33–38.

Liu, C., J. Han, and F. Lombardi (2014a). An analytical framework for evaluating the error characteristics of approximate adders. *IEEE Transactions on Computers (TC) 64*(5), 1268–1281.

Liu, C., J. Han, and F. Lombardi (2014b). A low-power, high-performance approximate multiplier with configurable partial error recovery. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–4.

Liu, G. and Z. Zhang (2017). Statistically certified approximate logic synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 344–351.

Liu, S., K. Pattabiraman, T. Moscibroda, and B. G. Zorn (2011). Flikker: Saving DRAM refresh-power through critical data partitioning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 213–224.

Liu, W., L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi (2017). Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on Computers (TC) 66*(8), 1435–1441.

Ma, J., S. Hashemi, and S. Reda (2021). Approximate logic synthesis using Boolean matrix factorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 41*(1), 15–28.

Mansley, C., A. Weinstein, and M. Littman (2011). Sample-based planning for continuous action markov decision processes. In *International Conference on Automated Planning and Scheduling (ICAPS)*, Volume 21, pp. 335–338.

Mcgeer, P., J. Sanghavi, R. Brayton, and A. S. Vincentelli (1993). ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 618–624.

Meng, C., W. Qian, and A. Mishchenko (2020). ALSRAC: Approximate logic synthesis by resubstitution with approximate care set. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 187:1–187:6.

Miao, J., A. Gerstlauer, and M. Orshansky (2013). Approximate logic synthesis under general error magnitude and frequency constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 779–786.

Miao, J., A. Gerstlauer, and M. Orshansky (2014). Multi-level approximate logic synthesis under general error constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 504–510.

Mishchenko, A. et al. (2022). ABC: a system for sequential synthesis and verification. `http://people.eecs.berkeley.edu/~alanmi/abc/`. Accessed September 1, 2022.

Mishchenko, A. and R. Brayton (2005). SAT-based complete don't-care computation for network optimization. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 412–417.

Mishchenko, A., R. Brayton, J.-H. R. Jiang, and S. Jang (2011). Scalable don't-care-based logic optimization and resynthesis. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 4*(4), 1–23.

Mishchenko, A., J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske (2006). Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 25*(5), 743–755.

Mittal, S. (2016). A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR) 48*(4), 1–33.

Mrazek, V. (2022). Optimization of BDD-based approximation error metrics calculations. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 86–91.

Mrazek, V. et al. (2022). EvoApproxLib: Library of approximate arithmetic circuits. https://github.com/ehw-fit/evoapproxlib/releases/tag/v1.2022/. Accessed September 1, 2022.

Mrazek, V., R. Hrbacek, Z. Vasicek, and L. Sekanina (2017). EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *IEEE/ACM Design, Automation & Test in Europe (DATE)*, pp. 258–261.

Mrazek, V., S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy (2016). Design of power-efficient approximate multipliers for approximate artificial neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7.

Mrazek, V., Z. Vasicek, and R. Hrbacek (2018). Role of circuit representation in evolutionary design of energy-efficient approximate circuits. *IET Computers & Digital Techniques 12*(4), 139–149.

Muller, J.-M. (1999). A few results on table-based methods. In *Developments in Reliable Computing*, pp. 279–288. Springer.

Nangate, Inc. (2022). Nangate 45nm open cell library. https://si2.org/open-cell-library/. Accessed October 1, 2022.

Nepal, K., Y. Li, R. I. Bahar, and S. Reda (2014). ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6.

Rahimi, A., A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta (2015). Approximate associative memristive memory for energy-efficient GPUs. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1497–1502.

Ranjan, A., A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan (2014). ASLAN: Synthesis of approximate sequential circuits. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6.

Ranjan, A., S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan (2015). Approximate storage for energy efficient spintronic memories. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6.

Rehman, S., W. El-Harouni, M. Shafique, A. Kumar, J. Henkel, and J. Henkel (2016). Architectural-space exploration of approximate multipliers. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8.

Roth, J. P. and R. M. Karp (1962). Minimization over Boolean graphs. *IBM Journal of Research and Development 6*(2), 227–238.

Russell, S. J. (2010). *Artificial intelligence a modern approach*. Pearson Education, Inc.

Saadat, H., H. Bokhari, and S. Parameswaran (2018). Minimally biased multipliers for approximate integer and floating-point multiplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 37*(11), 2623–2635.

Sampson, A., J. Nelson, K. Strauss, and L. Ceze (2014). Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS) 32*(3), 1–23.

Sapatnekar, S. (2004). *Timing*. Springer Science & Business Media.

Sasao, T. and J. T. Butler (2001). Worst and best irredundant sum-of-products expressions. *IEEE Transactions on Computers (TC) 50*(9), 935–948.

Scarabottolo, I., G. Ansaloni, G. A. Constantinides, and L. Pozzi (2021). A formal framework for maximum error estimation in approximate logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 41*(4), 840–853.

Scarabottolo, I., G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda (2020). Approximate logic synthesis: A survey. *Proceedings of the IEEE 108*(12), 2195–2213.

Scarabottolo, I., G. Ansaloni, and L. Pozzi (2018). Circuit carving: A methodology for the design of approximate hardware. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 545–550.

Schlachter, J., V. Camus, K. V. Palem, and C. Enz (2017). Design and applications of approximate circuits by gate-level pruning. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 25*(5), 1694–1702.

Schulte, M. J. and J. E. Stine (1997). Symmetric bipartite tables for accurate function approximation. In *IEEE International Symposium on Computer Arithmetic (ARITH)*, pp. 175–183.

Sentovich, E. et al. (1992). SIS: A system for sequential circuit synthesis. Technical report, University of California, Berkeley.

Shafique, M., W. Ahmad, R. Hafiz, and J. Henkel (2015). A low latency generic accuracy configurable adder. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6.

Shen, V.-S. and A. C. Mckellar (1970). An algorithm for the disjunctive decomposition of switching functions. *IEEE Transactions on Computers (TC) 100*(3), 239–248.

Shin, D. and S. Gupta (2011). A new circuit simplification method for error tolerant applications. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6.

Shoushtari, M., A. BanaiyanMofrad, and N. Dutt (2015). Exploiting partially-forgetful memories for approximate computing. *IEEE Embedded Systems Letters (ESL) 7*(1), 19–22.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature 529*(7587), 484–489.

Soeken, M., D. Große, A. Chandrasekharan, and R. Drechsler (2016). BDD minimization for approximate computing. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 474–479.

Stine, J. E. and M. J. Schulte (1999). The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology 21*, 167–177.

Su, S., C. Meng, F. Yang, X. Shen, L. Ni, W. Wu, Z. Wu, J. Zhao, and W. Qian (2022). VECBEE: A versatile efficiency–accuracy configurable batch error estimation method for greedy approximate logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 41*(11), 5085–5099.

Su, S., Y. Wu, and W. Qian (2018). Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 54:1–54:6.

Synopsys, Inc. (2022). Synopsys softwares. http://www.synopsys.com. Accessed October 1, 2022.

Tian, Y., T. Wang, Q. Zhang, and Q. Xu (2017). ApproxLUT: A novel approximate lookup table-based accelerator. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 438–443.

Vahdat, S., M. Kamal, A. Afzali-Kusha, and M. Pedram (2019). TOSAM: An energy-efficient truncation- and rounding-based scalable approximate multiplier. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 27*(5), 1161–1173.

Van Toan, N. and J.-G. Lee (2020). FPGA-based multi-level approximate multipliers for high-performance error-resilient applications. *IEEE Access 8*, 25481–25497.

Vasicek, Z. and L. Sekanina (2014). Evolutionary approach to approximate digital circuits design. *IEEE Transactions on Evolutionary Computation (TEVC) 19*(3), 432–444.

Venkataramani, S., K. Roy, and A. Raghunathan (2013). Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1367–1372.

Venkataramani, S., A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan (2012). SALSA: Systematic logic synthesis of approximate circuits. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 796–801.

Venkatesan, R., A. Agarwal, K. Roy, and A. Raghunathan (2011). MACACO: Modeling and analysis of circuits for approximate computing. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Waldrop, M. M. (2016). The chips are down for Moore's law. *Nature News 530*(7589), 144.

Witschen, L., T. Wiersema, M. Artmann, and M. Platzner (2022). MUSCAT: MUS-based circuit approximation technique. In *IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 172–177.

Wu, Y. and W. Qian (2016). An efficient method for multi-level approximate logic synthesis under error rate constraint. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 128:1–128:6.

Wu, Y., C. Shen, Y. Jia, and W. Qian (2017). Approximate logic synthesis for FPGA by wire removal and local function change. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 163–169. IEEE.

Xu, Q., T. Mytkowicz, and N. S. Kim (2015). Approximate computing: A survey. *IEEE Design & Test 33*(1), 8–22.

Yang, S. (1991). *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer.

Yao, Y., S. Huang, C. Wang, Y. Wu, and W. Qian (2017). Approximate disjoint bi-decomposition and its application to approximate logic synthesis. In *IEEE International Conference on Computer Design (ICCD)*, pp. 517–524.

Yazdanbakhsh, A., D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran (2016). AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test 34*(2), 60–68.

Ye, R., T. Wang, F. Yuan, R. Kumar, and Q. Xu (2013). On reconfiguration-oriented approximate adder design and its application. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 48–54.

Zhang, H.-T., J.-H. R. Jiang, L. Amarú, A. Mishchenko, and R. Brayton (2021). Deep integration of circuit simulator and SAT solver. In *ACM/IEEE Design Automation Conference (DAC)*, pp. 877–882.

Zhou, Z., Y. Yao, S. Huang, S. Su, C. Meng, and W. Qian (2018). DALS: delay-driven approximate logic synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7.

Zhu, N., W. L. Goh, and K. S. Yeo (2009). An enhanced low-power high-speed adder for error-tolerant application. In *IEEE International Symposium on Integrated Circuits (ISIC)*, pp. 69–72.

Zhu, N., W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong (2009). Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. *IEEE transactions on very large scale integration systems (TVLSI) 18*(8), 1225–1229.

Zhu, Q., N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli (2006). SAT sweeping with local observability don't-cares. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 229–234.

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：孟畅

日期：2023 年 5 月 10 日

上海交通大学

学位论文使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于 ☑公开论文

□内部论文，□1 年/□2 年/□3 年 解密后适用本授权书。

□秘密论文，____年（不超过 10 年）解密后适用本授权书。

□机密论文，____年（不超过 20 年）解密后适用本授权书。

（请在以上方框内打“√”）

学位论文作者签名：孟畅　　指导教师签名：钱炜懋

日期：2023 年 5 月 10 日　　日期：2023 年 5 月 11 日

20006301

上海交通大学博士学位论文答辩决议书

020370210001

姓　名	孟畅	学号	020370210001	所在学科	电子科学与技术
指导教师	钱炜慷	答辩日期	2023-05-17	答辩地点	密西根学院龙宾楼414B
论文题目	Efficient Synthesis Methods for High-Quality Approximate Computing Circuits and Architectures				

投票表决结果：5 /5 /5（同意票数/实到委员数/应到委员数） 答辩结论：■通过 □未通过

评语和决议：

本论文研究了面向近似计算电路和架构的高效综合方法，为后摩尔时代提高芯片质量探索了可能的技术路径。作者在电路层和架构层两个层次提出了高效的综合方法，以降低芯片的面积、延时和功耗，并缩短近似计算芯片的开发周期。主要创新性成果如下：

1) 提出了优化电路面积的近似逻辑综合方法，其中包括基于“近似重替换”的局部近似变换（LAC）方法以及有效的 LAC 顺序调度方法。实验结果表明，综合后的电路面积得到了较大的减小。

2) 提出了优化电路延时的近似逻辑综合方法，该方法提出了“关键误差图”这一数据结构，同时缩短了所有关键路径，实现了电路延时的降低。

3) 提出了快速的最大误差检查算法，通过“偏布尔差分”技术来进行建模，实现了快速的最大误差检查，缩短了面向最大误差约束的近似逻辑综合流程的运行时间。

4) 提出了低功耗、低延迟的近似查找表架构，并针对该架构，设计了基于整数线性规划和启发式方法的高效查找表配置算法，降低了近似查找表架构的能耗。

论文组织合理，叙述清晰。答辩中作者出色地回答了委员们提出的问题。论文答辩委员会认为，论文达到了国家学位条例对博士学位论文的要求，且该论文是一篇优秀的博士论文。作者具有坚实宽广的基础理论和系统深入的专门知识，具备独立从事科研工作的能力。论文答辩委员会5人投票，5票赞成、0票弃权、0票反对。根据投票结果，论文答辩委员会一致同意孟畅同学通过博士学位论文答辩，同意其毕业，并建议授予工学博士学位。

2023 年 5月 17日

答辩委员会成员签名	职务	姓名	职称	单位	签名
	主席	钱炜慷	副教授	上海交通大学上海交大密西根学院	钱炜慷
	委员	PAUL AN-LIN	副教授	上海交通大学上海交大密西根学院	
	委员	邹桉	讲师	上海交通大学上海交大密西根学院	邹桉
	委员	纪志罡	教授	上海交通大学电子信息与电气工程学院（微纳电子学系）	
	委员	姜红兰	副教授	上海交通大学电子信息与电气工程学院（微纳电子学系）	姜红兰