UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL AMMES PINHO

A Two-Level Approximate Logic Synthesis Method Based on Insertion and Removal of Cubes

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Renato Perez Ribas

Porto Alegre
July 2021

CIP — CATALOGING-IN-PUBLICATION

Pinho, Gabriel Ammes

A Two-Level Approximate Logic Synthesis Method Based on Insertion and Removal of Cubes / Gabriel Ammes Pinho. – Porto Alegre: PPGC da UFRGS, 2021.

106 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Renato Perez Ribas.

1. Approximate computing. 2. Approximate logic synthesis. 3. Literal count. 4. Two-level circuit. I. Ribas, Renato Perez. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Prof[a]. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof[a]. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"It's okay to lose your way... just don't lose sight of what you have decided."

— Roronoa Zoro (Eiichiro Oda)

ACKNOWLEDGMENT

Inicio os agradecimentos pela minha mãe, Tanira, e meu dois pais, Nauro e Silvio, que mesmo estando longe sempre me apoiaram nos diversos desafios enfrentados durante minha formação. Continuando, agradeço a minha namorada Elisandra, que sempre esteve comigo durante esta jornada, sempre foi uma companheira para todas as horas e aguentou minhas atividades acadêmicas madrugada a dentro, esta inclusive, e aos meus sogros, Nilda e Dalvo, que me acolheram como filho desde o dia que os conheci.

Gostaria de agradecer muito aos professores Renato Ribas e Paulo Butzen e ao Walter Lau Neto (Casão), pelas reuniões semanais durante o desenvolvimento remoto deste trabalho, sugerindo temas e abordagens, ajudando na elaboração e correção de trabalhos (este inclusive). Tenho certeza que sem estas reuniões este trabalho não chegaria nem perto do que se tornou.

Agradeço a todos que mantiveram comigo conversas e discussões dos diversos assuntos, principalmente durante esta etapa remota, entre estes, Ribas, Butzen e Casão citados anteriormente; aos amigos, Adão, Andre, Chico, Duda, Matheus, Tibu e Zancan; e à Elisandra, além dos tantos outros anos, compartilhou comigo também estes últimos anos de isolamento social.

Por fim, agradeço aos aos membros da banca, os professores José Rodrigo Furlanetto Azambuja e Leomar Soares da Rosa Junior e o Dr. Felipe dos Santos Marranghello, por terem aceitado o convite de avaliar este trabalho e pelas dicas valiosas dadas na correção.

ABSTRACT

Approximate computing is a design paradigm that allows systems to have imprecise or inexact execution, aiming to optimize performance and power dissipation. When approximate computing is applied to systems that perform error-resilient applications, it is possible to optimize without critically degrading the expected application operation. The presented work focuses on exploring approximate computing at the circuit level, more specifically, digital integrated circuits (IC). Electronic design automation tools provide a highly automated IC design flow, which may be roughly divided into three main steps: high-level synthesis, logic synthesis and physical synthesis. The logic synthesis step has as a goal the gate network building and optimization targeting a given technology. The logic synthesis is performed over two-level (2L) or multiple-level topologies of combinational blocks. The employment of approximate computing at the circuit level consists of deriving a gate network implementation that is not logically equivalent to the original circuit behavior specification but can improve area, performance and power dissipation. Several works propose techniques to approximate circuits automatically by systematically modifying a general logic behavior without exceeding a given error threshold. Due to the similarity among those techniques, the adopted data structures and optimization goals through logic synthesis, the automatic construction of approximate circuits at this design phase is called *approximate logic synthesis* (ALS). Conventional logic synthesis methods to build two-level circuits are employed as part of multilevel synthesis methods and for synthesizing systems over CPLD architectures. Besides the benefits of approximate sum-of-products (SOP) and product-of-sums (POS), the approximation of 2L circuits can be exploited in both applications. Moreover, investigations over 2L-ALS methods represent an important contribution to further advancements also on multilevel ALS. This work proposes a two-level approximate logic synthesis method that receives an SOP expression and a given error rate threshold as inputs and generates an approximate SOP with an optimized number of literals. In this work, we also intend to derive a scalable method that allows the insertion of more errors than is observed in existing 2L-ALS works. The experimental results show that the proposed approach can derive SOPs with fewer literals compared to the state-of-the-art method for the same amount of errors. The obtained solutions reach an average literal count reduction of circa 38% with an error rate of 1%, 56% with an error rate of 3%, and up to 93% with an error rate of 5%. Moreover, as it is unknown how approximations made in two-level structures impact the quality of mul-

tilevel circuit design, we have carried out some experiments that apply two-level, and multilevel ALS approaches interchangeably to build approximate circuits and so analyze the obtained solutions.

Keywords: Approximate computing. approximate logic synthesis. literal count. two-level circuit.

Método de Síntese Lógica Aproximada Dois-Níveis Baseado na Inserção e Remoção de Cubos.

RESUMO

Computação aproximada é um paradigma que permite que um sistema tenha uma execução imprecisa ou inexata com o objetivo de otimizar o seu desempenho e sua eficiência energética. Quando este paradigma é aplicado em sistemas que executam funções resilientes a erros, é possível otimizar o sistema sem degradar de forma crítica a operação desejada. Este trabalho foca no uso de computação aproximada no nível de circuitos, em particular, nos circuitos integrados digitais (CI). Ferramentas de projeto de circuitos integrados fornecem um fluxo altamente automatizado para do projeto de CIs. Este fluxo pode ser dividido em três passos principais: síntese de alto nível, síntese lógica e síntese física. A síntese lógica tem como objetivo otimizar a lógica do circuito e implementá-lo em uma dada tecnologia alvo. A síntese lógica é executada sobre representações dois-níveis ou multinível que implementam a lógica combinacional de um dado circuito. A aplicação de computação aproximada no nível de circuitos consiste em obter uma implementação que não é logicamente equivalente à especificação mas consegue realizar otimizações em área, desempenho e consumo de energia. Diversos trabalhos propõem técnicas para aproximar um circuito de forma automática através de modificação sistemática do funcionamento de um circuito genérico sem exceder uma dada restrição de erro. Devido à similaridade em técnicas, estruturas de dados e objetivos de otimização com a etapa de síntese lógica, a geração automática de circuitos aproximados é frequentemente chamada de síntese lógica aproximada. Métodos de síntese lógica tradicional para construção de circuitos dois-níveis podem ser utilizados para síntese de componentes programáveis CPLDs, bem como parte de métodos para síntese de circuitos multinível. Além da geração de expressões aproximadas do tipo somas-de-produtos (SOP) e produtos-de-somas (POS), técnicas de aproximação para circuitos dois-níveis poder ser exploradas nesses dois canários. Além disso, o entendimento dos conceitos e técnicas relacionados à aproximação dois-níveis pode contribuir significativamente para futuros estudos sobre a aproximação de circuitos multinível. Este trabalho propõe um método para aproximar circuitos dois-níveis que tem como entrada uma SOP e um dado limite de frequência de erro, e gera uma expressão aproximada com um número de literais reduzido e que respeita o dado limite de erro. O método proposto foi desenvolvido com a intenção de ser escalável

em relação à quantidade de erros permitidos, possibilitando uma inserção de mais erros do que é observado em outros trabalhos que abordam o mesmo problema. Nos resultados experimentais obtidos, o método proposto gerou SOPs aproximadas com menos literais do que o obtido pelo método considerado estado-da-arte nesta tecnologia, para a mesma quantidade de erros. Comparando com a SOP original, o presente método obteve uma redução de literais média de 38% com frequência de erro de 1%, 56% com frequência de erro de 3%, e de até 93% frequência de erro de 5%. Como o impacto das aproximações feitas em uma representação dois-níveis na qualidade de um circuito multinível, e vice-versa, é desconhecido, foram feitos experimentos explorando técnicas de aproximação dois-níveis e multinível em conjunto para geração de circuitos aproximados, e para análise das soluções obtidas.

Palavras-chave: Computação aproximada, síntese lógica aproximada, número de literais, lógica dois-níveis..

LIST OF ABBREVIATIONS AND ACRONYMS

ADD	Algebraic Decision Diagrams
AIG	And-Inverter Graph
ALS	Approximate Logics Synthesis
ASIC	Application-specific integrated circuit
BDD	Binary Decision Diagram
CNF	Conjunctive Normal Form
COM	Cover Once Minterm
CPLD	Complex Programmable Logic Device
CPU	Central Process Unit
CSBF	Completely Specified Boolean Function
DAG	Direct Acycle Graph
EDA	Electronic Design Automation
EIC	Erronous Input Combination
ER	Error Rate
FA	Full-Adder
FPGA	Field Programmable Gate Array
GPU	Graphic Process Unit
IC	Integrated Circuit
IoT	Internet of Things
ISA	Instruction Set Architecture
ISBF	Incompletely Specified Boolean Function
LUT	LookUp Table
MAE	Mean Absolute Error
MRE	Mean Relative Error

MSE	Mean Square Error
NoE	Number of Errors
NPU	Neural Process Unit
NVM	Non-Volatile Memory
PAL	Programmable Array Logic
PI	Primary Input
PLA	Programmable Logic Array
PO	Primary Output
POS	Product-of-Sums
RAM	Random Accedd Memory
RMS	Recognition, Mining and Synthesis
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	register transfer level
SAT	Satisfiability
SCA	Symbolic Computer Algebra
SCT	SICC-cube tree
SICC	Set of Input Combinations for 0-to-1 output Complement
SOP	Sum-of-Products
WBF	Worst Bit-Flip Error
WCE	Worst Case Error
WRE	Worst Relative Error

LIST OF FIGURES

Figure 2.1 PLA format representation of the function described in Table 2.1................25
Figure 2.2 PLA architecture configuration for the function presented in Table 2.1. 26
Figure 2.3 BDD representation of the function described in Table 2.1. 26
Figure 2.4 Optimized ROBDD of the function described in Table 2.1...........................27
Figure 2.5 Two Boolean network representations of the function described in Table 2.1..28
Figure 2.6 AIG representation of the function described in Table 2.1............................29
Figure 2.7 Karnaugh map representation of the function described in Table 2.1. 30
Figure 2.8 Karnaugh map with all possible cubes highlighted......................................30
Figure 2.9 Karnaugh map containing the SOP $!x_0*!x_1 + x_0 * x_1 + x_0 * x_2$ 31

Figure 3.1 Example of two miter structures..47
Figure 3.2 Karnaugh map containing the cover used on examples. The SOP represented by this Karnaugh map is $!x_0*!x_1 + x_0 * x_1 + x_0 * x_2$ 50
Figure 3.3 Example of an approximation done by a cube insertion technique. 51
Figure 3.4 Example of an approximation done by a cube removal technique................51
Figure 3.5 Hasse diagram for a function with 2 inputs and 2 outputs. 52
Figure 3.6 Hasse diagram of 3-inputs 1-output function with the cubes that can be enumerated by the exhaustive technique when using $\mathcal{C}$ as the original function highlighted in green. 53
Figure 3.7 Hasse diagram of 3-inputs 1-output function with the cubes that an expansion technique can enumerate over the cubes in $\mathcal{C}$ highlighted in blue.............54

Figure 4.1 Approximate covers examples comparing cube insertion, cube removal and both approaches together. 57
Figure 4.2 The behavior of the iterative partial solutions generation, considering a limit of 4 NoE. Each partial solution generates two new partial solutions, increasing NoE in 1 and 2 until it reaches the NoE limit.......................................61
Figure 4.3 Graph showing the number of solution generated with each NoE for an approximation with a limited to 8 NoE, totalizing 87 partial solutions. 62

Figure A.1 Exemplos de coberturas aproximadas comparando o uso da inserção de cubos, da remoção de cubos e de ambas em conjunto..99

LIST OF TABLES

Table 2.1 Example of a truth table for a 3-inputs CSBF. 22
Table 2.2 Example of a truth table for a 3-inputs 2-outputs CSBF. 22
Table 2.3 Example of a truth table for a 3-inputs 2-outputs ISBF. 22
Table 2.4 Example of finding all prime implicants of the function described in Table 2.1 31

Table 3.1 Example of Boolean relation. 48
Table 3.2 Example of converting a 2-bit adder with WCE threshold of 1 in a Boolean relation. 49

Table 5.1 Comparison to Su's approach (SU et al., 2020) in IWLS'93 benchmark suite considering NoE threshold of 16. 74
Table 5.2 Results from proposed method considering ER threshold in IWLS'93 benchmark suit. 77
Table 5.3 Post Processing 79

Table 6.1 Two-Level approximation impact on multilevel circuits 84

Table A.1 Comparação do método proposto com o método de Su (SU et al., 2020) sobre os circuitos do conjunto de benchmarks do IWLS93 considerando ER como métrica de erro 104
Table A.2 Resultados do método proposto sobre os circuitos do conjunto de benchmarks do IWLS93 considerando ER como métrica de erro 105

CONTENTS

1 INTRODUCTION...15
1.1 Motivation and Proposed Work...17
1.2 Text Organization 19
2 PRELIMINARIES...20
2.1 Boolean Function Definition 20
2.2 Boolean Function Representation 21
2.2.1 Truth Table...21
2.2.2 Boolean Expression 23
2.2.3 Programmable Logic Array 24
2.2.4 Binary Decision Diagram 25
2.2.5 Boolean Networks...27
2.2.6 AND-Inverter Graph...28
2.3 Two-level Logic Optimization...29
2.4 Boolean Satisfiability 33
2.5 Espresso Tool...33
3 APPROXIMATE COMPUTING PARADIGM 35
3.1 Error-Resilient Application...35
3.2 Approximate Computing...36
3.3 Approximate Circuits 37
3.4 Approximate Logic Synthesis 39
3.5 Error Modeling 42
3.5.1 Error Metrics...42
3.5.2 Error Calculation Methods...44
3.6 Two-Level Approximate Logic Synthesis...48
3.6.1 ER and WCE Bounded Methods 48
3.6.2 ER Bounded Methods...50
4 PROPOSED 2L-ALS METHOD...56
4.1 Su's Cube Insertion Approach...56
4.2 Data Structure...59
4.2.1 Cover...59
4.2.2 SICC-cube tree (SCT)...60
4.2.3 Solutions 60
4.3 General Description...60
4.4 Cube Insertion Procedure 63
4.4.1 SICC Cube-Tree Generation...64
4.4.2 Combine and Estimate 65
4.5 Cube Removal Procedure...67
4.5.1 Speed-Up Optimization 69
4.6 Post-Processing Tools...71
4.7 Time Complexity Analysis 72
5 EXPERIMENTAL RESULTS 73
5.1 Comparison to the State-of-Art Approach 73
5.2 Insights About the Runtime Reduction 74
5.3 Results with Error Rate...76
5.4 Post-Processing Result...78
6 MULTILEVEL COMPARISON 81
7 CONCLUSION 87
REFERENCES...88

APPENDIX A — RESUMO DA DISSERTAÇÃO **94**
A.1 Introdução **94**
A.1.1 Motivação e Proposta 94
A.2 Conceitos Preliminares **95**
A.3 Estado-da-Arte **96**
A.4 Trabalho Proposto **98**
A.4.1 Descrição Geral 99
A.4.2 Método de Inserção de Cubos 100
A.4.3 Função de Remoção de Cubos 101
A.5 Resultados Experimentais **102**
A.5.1 Comparação com o Estado-da-Arte 103
A.5.2 Resultados Considerando a Frequência de Erro 104
A.6 Conclusão **106**

1 INTRODUCTION

In the current world, applications that use digital signal processing, multimedia processing, WEB search, data analytics, RMS (recognition, mining and synthesis), machine learning, and sensors, as Internet-of-Things (IoT) applications, are increasingly present in our live. These applications have a common characteristic: they are error-resilient applications. An error-resilient application is an application that can produce acceptable results even with errors happening during its execution. For instance, those errors can occur by hardware faults or noisy data in the input signals (CHIPPA et al., 2013).

On the other hand, approximate computing is a paradigm that allows a system to have an imprecise or inexact execution, aiming to optimize performance and power dissipation. When approximate computing is applied to systems that perform error-resilient applications, it is possible to optimize the system without critically degrading the expected application operation (HAN; ORSHANSKY, 2013; MITTAL, 2016; XU; MYTKOWICZ; KIM, 2016).

The concept of approximate computing can be applied in different computational levels from the higher abstraction views, like algorithms and compilers, passing through the architectural and reaching the circuit level. The presented work focuses on the application of approximate computing at the circuit level.

The circuit level of computer designs is composed of integrated circuits (IC). Since the beginning of ICs fabrication in 1958, the number of components per area in an IC has doubled every 18 months (WESTE; HARRIS, 2010). In the beginning, the IC design has been handmade by the designer. With the continuous increase in the circuit design complexity, this approach became impracticable (RABAEY; CHANDRAKASAN; NIKOLIC, 2002). Computer-aided design (CAD) tools started to be used to handle the crescent number of components. Such CAD tools compose electronic design automation (EDA) environments, indispensable for any modern electronic system.

Currently, several CAD tools are applied at different phases of the IC design and organized inside EDA frameworks. The ASIC (application-specific integrated circuit) design flow can be roughly divided into three main phases (MICHELI, 1994):

- *High-level synthesis* is responsible for translating the circuit behavior, described algorithmically as in System C language, into a hardware description format, like register transfer level (RTL), which represents the circuit behavior at the architec-

tural level.

- *Logic synthesis* comprises the circuit logic building at the gate level, usually focusing on optimizing area, performance, and power consumption in a target technology.
- *Physical synthesis* assigns physical resources to specific positions on the chip and routes internal interconnection wire aiming the final fabrication.

The presented work is related to the logic synthesis phase. This phase consist of three main steps (MICHELI, 1994):

- *Technology-independent optimization* step aims to optimize the hardware description using generic Boolean data structures, abstracting physical characteristics of the final circuit technology.
- *Technology mapping* step binds the optimized Boolean structure to the logic cells containing details about the circuit physical characteristics.
- *Technology-dependent optimization* step optimizes the mapped circuit considering its physical building characteristics.

Moreover, a circuit can be seen comprising sequential and combinational blocks (MICHELI, 1994). In this sense, the sequential synthesis deals with the register elements on the design, whereas the combinational synthesis aims to optimize the existing logic operations between sequential elements. In this work, we focus on the combinational synthesis field. One of the main tasks of the technology-independent optimization step over combinational circuits is to find a representation in a Boolean data structure that is logically equivalent to the circuit specification that optimizes metrics related to area and performance.

Furthermore, the circuit construction can be done at two levels or multi levels (three or more) of logic depth. Two-level (2L) circuits are usually composed of two logic operations divided into two layers. The most used two-level representations are sum-of-products (SOP) and products-of-sums (POS). Multilevel circuits do not have a limitation related to a specific logic operation structure. There are several Boolean structures to represent multilevel circuits. One of the most used multilevel representations is Boolean networks, which uses directed acyclic graph (DAG) to represent the logic operations and their interconnections. The two-level circuit optimization focuses on reducing the number of operations in SOP/POS expressions. The multilevel optimization objective may vary according to the used Boolean structure, but, in general, it aims to reduce the number of

operations and levels.

The application of approximate computing at circuit level consists of deriving a circuit implementation that is not logically equivalent to the original circuit behavior specification but can present further optimizations in area, performance and power dissipation (SCARABOTTOLO et al., 2020). This approximate implementation is called an *approximate circuit*. In that way, techniques of technology-independent optimization and approximate circuit generation address related problems. The major difference between these techniques is the possibility of deriving a final circuit implementation that is not equivalent to the specification.

The first works on approximate circuit approximation focused on manually approximating the target design. Such a handcrafted approximation demands the circuit designer to have deep knowledge about the circuit behavior to make approximations that optimize the circuit but do not degrade the output quality-of-results (QoR). The main application of manual approximation has been arithmetic designs because those circuit regular structures are extensively studied in literature and are widely used in multiple designs (JIANG; HAN; LOMBARDI, 2015; JIANG et al., 2016). The main drawback of this approach is the required expertise on the circuit behavior to approximate it, turning it into a design-specific technique. The automatic circuit approximation approach aims to overcome these drawbacks.

The automatic circuit approximation consists of automatically modifying a general circuit logic behavior without knowing implementation details. It is done by systematically modifying the circuit logic behavior without exceeding a given error threshold. Due to the similarity in techniques, data structures, and optimization objectives with technology-independent optimization techniques, the automatic generation of approximate circuits is commonly called *approximate logic synthesis* (ALS).

1.1 Motivation and Proposed Work

Methods to synthesize 2L circuits are employed as part of multilevel synthesis methods (UMANS; VILLA; SANGIOVANNI-VINCENTELLI, 2006) and also for synthesizing logic over CPLDs architectures (KUBICA; KANIA, 2019). Besides the approximate SOP/POS generation, the approximation of 2L circuit techniques can be used in both applications. Moreover, understanding techniques and concepts of 2L-ALS methods represent an important advancement for future investigation on multilevel ALS.

The works that address the 2L-ALS problem focus on approximate an SOP expression (SHIN; GUPTA, 2010; MIAO; GERSTLAUER; ORSHANSKY, 2013; ZOU; QIAN; HAN, 2015; SU et al., 2020). The optimization of 2L circuits focuses on reducing the number of cubes or the number of literals. In general, 2L-ALS tools focus on literal reduction. There ate two main ways to perform it: the insertion and removal of SOP cubes (SHIN; GUPTA, 2010). The first one inserts new cubes that could not be included without resulting in errors. If the cube insertion is done cleverly, it implies removing some cubes and optimizing the SOP expression in terms of literal count. The second one consists of directly removing cubes from the SOP without previously inserting a new cube. The optimization on the SOP is proportional to the number of operations of the removed cube, i.e., to the reduced number of literals.

In (SHIN; GUPTA, 2010), the authors present some experiments suggesting that the cube insertion technique results are better than the cube removal procedure. Based on this experiment, the subsequent works presented in the literature have focused just on the cube insertion technique (ZOU; QIAN; HAN, 2015; SU et al., 2020).

This work proposes a two-level approximate logic synthesis method that receives an SOP and generates the approximate SOP with an optimized number of literals respecting a given error rate threshold. To perform that, we apply cube insertion and removal procedures together. Such a strategy leads to better or equal solutions when compared to the only cube insertion technique, as done by other approaches, without a significant run-time penalty.

A bottleneck observed on existing 2L-ALS methods is the scalability when the quantity of allowed error increase. In this work, we intend to derive a scalable method with the capability to insert more errors than other 2L-ALS approaches. For instance, in (SHIN; GUPTA, 2010) and in (SU et al., 2020), the authors limit the number of allowed errors to 8 and 16 due to the prohibitive execution time. The development of a scalable method allows us to derive an approximate circuit using a percentage error rate threshold, as many approximate circuits works use. Using a percentage error threshold implies that the number of allowed errors depends on the number of input combinations in a circuit instead of instead of a fixed error number.

As mentioned, methods of 2L logic synthesis can be exploited as part of the multilevel circuit synthesis. Currently, it is unknown how approximations made over 2L representation impact the QoR of multilevel topology. To understand the relationship of 2L and multilevel approximations impact each other, we propose an experiment that applies both

ALS approaches together to generate approximate circuits and evaluate the solutions.

1.2 Text Organization

The remaining of this manuscript is organized as follows:

Chapter 2 reviews the background concepts for a better understanding of the proposed method. There are basic definitions and nomenclatures used in Boolean functions and the main Boolean structures. It is also presented some details about synthesizing an SOP expression.

Chapter 3 presents a broad review of the approximate computing paradigm and related works. It begins by presenting the concepts of error-resilient applications and approximate computing. A quick review of works that applies approximate computing over different computational levels is also shown. Next, it focuses on approximate circuits and how to generate them, starting from the handcraft approximation works until the ALS. At first, are presented a list of multilevel ALS approaches, with some details of each one. Then, we discuss how the error inserted during the circuit approximation is calculated, linking to the presented ALS works. At the end, a detailed explanation of 2L-ALS approaches is given, relating it to the proposed method.

Chapter 4 describes the proposed 2L-ALS method. It starts given the main idea of our approach. Then, the data structures used to implement the method are shown. The general flow of the method is presented, followed by details about the cube insertion and the cube removal procedures. At the end, some post-processing techniques and time complexity analysis are presented.

Chapter 5 presents the experimental results. This chapter compares our solutions to the ones obtained by the state-of-art approach, and shows how our method behaves with a percentual error threshold. It is also presented some insights about the runtime and a discussion about the post-processing tools.

Chapter 6 presents the experiments to analyze the impact of using the proposed 2l-ALS method to approximate multilevel circuits. It also presented results of using 2L and multilevel ALS methods together to generate approximate circuits.

Finally, in Chapter 7, the contributions of this work are summarized and we conclude with final considerations.

2 PRELIMINARIES

This chapter introduces the adopted notation and preliminaries useful for a better understanding of this work. It gives the reader a brief description of Boolean functions, Boolean structures, and synthesis of 2L circuits.

2.1 Boolean Function Definition

An n-input *Boolean function* f(X) defined by the variable (support) set $X = \{x_0, ..., x_{n-1}\}$ is a mapping

$$f(X) : \mathbb{B}^n \rightarrow \mathbb{B}$$

where $\mathbb{B}$ = {0, 1} and $n \in \mathbb{N}$. This Boolean function is also defined as a completely specified Boolean function (CSBF). An element $m \in B^n$, i.e., an n-bit vector is called *minterm*. There are 2^n minterms in B^n. The on-set of function f comprises all minterms m such that f(m) = 1 and is denoted ON-SET(f). Conversely, the set representing all minterms such that f(m) = 0 is called the off-set and is denoted OFF-SET(f). Notice that a CSBF can be uniquely represented by its on-set or off-set. A constant function one (1) has an empty off-set, while the constant function zero (0) contains no element in the on-set.

A multiple-output Boolean function f(X), defined by the variable set X, is a mapping

$$f(X) : \mathbb{B}^n \rightarrow \mathbb{B}^m$$

where *n* and *m* $\in \mathbb{N}$. Each minterm is mapped to an m-bit vector o $\in \mathbb{B}^m$. An n-input m-output Boolean function can be split into m n-input Boolean functions, each one with its on-set and off-set.

An n-input incompletely specified Boolean function (ISBF) f(X), defined by the variable set X, is a mapping

$$f(X) : \mathbb{B}^n \rightarrow \mathbb{Y}$$

where $\mathbb{Y}$ = {0, 1, -}. Notice that ISBF differs from CSBF in the fact that the former may also assume *don't care* (-) values, besides the binary values *0* and *1* (BRAYTON et

al., 1984). Besides ON-SET(f) and OFF-SET(f), an ISBF also contains a DC-SET(f), which comprises all minterms m such that f(m) = -. An ISBF can be uniquely represented by a pair of its on-set, off-set and dc-set. An multiple-output ISBF f(X), defined by the variable set X, is a mapping

$$f(X) : \mathbb{B}^n \rightarrow \mathbb{Y}^m$$

where n and $m \in \mathbb{N}$ and each minterm is mapped to an m-bit vector o $\in \mathbb{Y}^m$.

In this work, the terms "function" and "Boolean function" are used interchangeably, unless otherwise stated.

The relationship between the domain and the image sets of a CSBF is usually done by logic operations over the variables from X. There are some basic operators, such as AND ($\wedge$ or *), OR ($\vee$ or +) and NOT ($\neg$ or !). The AND operator evaluates the function to 1 when all n variables are true (i.e., equal to 1). Otherwise, the function is evaluated to 0. The OR operator, in turn, evaluates to 0 when all variables are false (i.e., equal to 0). Otherwise, the function is evaluated to 1. The NOT operator, also known as negation, performs over only a single variable and returns 0 if the variable is 1, and vice-versa. As this work focuses on CSBF, it is not defined operations over ISBF.

2.2 Boolean Function Representation

There are several ways to represent Boolean functions, differing in the trade-off between simplicity and scalability. Therefore, this section presents an overview of structures and formats for representing Boolean functions used or referred to in this work.

2.2.1 Truth Table

The most straightforward way of representing functions is the truth table. In this representation, the output value of a function is specified for each possible combination of Boolean values assigned to the input variables. For a function with n inputs, the truth table is composed of 2^n rows and, consequently, by 2^n minterms, which are the product over the variables. For instance, let the function $f(x_0, x_1, x_2)$ be represented by the truth table shown in Table 2.1. The minterms $!x_0*!x_1*!x_2$, $!x_0*!x_1*x_2$, $x_0*!x_1*x_2$, $x_0*x_1*!x_2$

and $x_0 * x_1 * x_2$ are in the on-set of f while $!x_0 * x_1 *!x_2$, $!x_0 * x_1 * x_2$ and $x_0 *!x_1 *!x_2$ are in the off-set. For the rest of this work, we refer to the minterms on the on-set of a function as minterms for simplicity. A truth table can also represent multiple-output completely and incompletely specified functions. In Figure 2.2 is shown the truth table of a 3-input 2-output CSBF, and the truth table of a 3-input 2-output ISBF is illustrated in Figure 2.3.

Table 2.1 – Example of a truth table for a 3-inputs CSBF.

x_0	x_1	x_2	f	minterm
0	0	0	1	$!x_0 *!x_1 *!x_2$
0	0	1	1	$!x_0 *!x_1 * x_2$
0	1	0	0	$!x_0 * x_1 *!x_2$
0	1	1	0	$!x_0 * x_1 * x_2$
1	0	0	0	$x_0 *!x_1 *!x_2$
1	0	1	1	$x_0 *!x_1 * x_2$
1	1	0	1	$x_0 * x_1 *!x_2$
1	1	1	1	$x_0 * x_1 * x_2$

Table 2.2 – Example of a truth table for a 3-inputs 2-outputs CSBF.

x_0	x_1	x_2	y_0	y_1
0	0	0	1	1
0	0	1	1	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Table 2.3 – Example of a truth table for a 3-inputs 2-outputs ISBF.

x_0	x_1	x_2	y_0	y_1
0	0	0	1	-
0	0	1	1	-
0	1	0	0	0
0	1	1	-	0
1	0	0	-	-
1	0	1	1	1
1	1	0	-	1
1	1	1	1	1

2.2.2 Boolean Expression

Another way to represent a CSBF is through a mathematical expression. Such an expression consists of a mathematical formula comprising Boolean operators and input variables. Each input variables instance is called a *literal*. The expression *logic depth*, in turn, consists of the number of nested Boolean operations, disregarding the NOT operation. In general, literals and logic depth are quality metrics for Boolean expressions, and usually, it aims to make these values as small as possible.

Boolean expressions have two main classifications: two-level (2L) expressions and multilevel expressions.

A 2L expression, as the name suggests, limits the expression logic depth in two. Sum-of-products (SOP) and product-of-sums(POS) are examples of 2L expressions. Both patterns use an operation over the literals and another one to link them. The SOP (POS) expression uses ANDs (ORs) over the inputs and ORs (ANDs) to link. In an SOP, a set of literals linked by an AND operator is called a *cube*. This work focuses on SOP expressions. This SOP is canonical when all cubes of an SOP represent all minterms on the ON-SET(f). However, it is usually of interest to represent the SOP of a given function with the minimal number of cubes and literals.

The canonical SOP representation for the function described in Table 2.1 is

$$!x_0 * !x_1 * !x_2 + !x_0 * !x_1 * x_2 + x_0 * !x_1 * x_2 + x_0 * x_1 * !x_2 + x_0 * x_1 * x_2,$$

while the SOP for the same function with the minimal number of literals is

$$!x_0 * !x_1 + x_0 * x_1 + x_0 * x_2.$$

A set of cubes covering all ON-SET(f) minterms and not covering any OFF-SET(f) minterms is called a *cover* for the function f. So, an SOP is a way to represent a cover of a given function.

An *implicant* is a cube the covers one or more ON-SET(f) minterms and do not cover any OFF-SET(f) minterms. A *prime implicant* is an implicant that is not covered by any other implicant of the function. An *essential prime implicant* is a prime implicant covering at least one ON-SET(f) minterm that no other prime implicant covers. An *irredundant cover* is a cover where removing any cube implies at least one ON-SET(f) minterm uncovered. An irredundant cover only containing prime implicants is also called

an *ISOP*.

Using some of the previous definitions to represent a multiple-output function is possible. Still, it is necessary to modify the literal, minterm, and cube definitions to handle multiple-output functions. Besides an instance of an input variable, the literal definition must also encompass the output variables. So, there are two types of literals: the input literals, representing an instance of an input variable, and the output literal, representing an instance of an output variable. For a cube to be used in a multiple-output function representation, the output literals must also be used in the product. In that way, if the output y_i is in the cube, the i-th output is just 1 when the cube evaluates to 1. Otherwise, the i-th output does not depend on the cube value. With this notation, a minterm contains all input literals and only one output literal. As an SOP represent a single-output function, an m-output function can be represented as a set of *m* SOPs, where each one represents an output. For example, the first line of the truth table in Table 2.2 presents two minterms, with the minterm $y_0*!x_0*!x_1*!x_2$ related to the output y_0 and $y_1*!x_0*!x_1*!x_2$ related to the output y_1. The cube $y_0 * y_1*!x_0*!x_1*!x_2$ represents both minterms. Taking into account this notation, the multiple-output cover for the function described in Table 2.2 is

$$\{y_0 * y_1*!x_0*!x_1*!x_2, y_0*!x_0*!x_1 * x_2, y_1 * x_0*!x_1*!x_2, y_0 * y_1 * x_0*!x_1 * x_2,$$

$$y_0 * y_1 * x_0 * x_1*!x_2, y_0 * y_1 * x_0 * x_1 * x_2\}.$$

A multilevel expression does not have any limitation regarding its logic depth. Thus, these expressions do not have a strict structure, and it is possible to nest multiple Boolean operations. In general, a multilevel expression of f decreases the number of literals at the cost of increasing the logic depth compared to a two-level expression of f. A multilevel expression representing the function described in Table 2.1 is

$$!x_0*!x_1 + x_0 * (x_1 + x_2).$$

2.2.3 Programmable Logic Array

The programmable logic array (PLA) format was thought to describe circuits implemented in related AND-OR structure(CHEN, 2006). This technology consists of a fixed configurable architecture, which can be programmed to compute combinational

logic. The architecture comprises an array of AND logic gates connected to the circuit inputs, followed by an array of OR logic gates connected to the circuit outputs. In PLAs structure, both arrays are programmable, differing from other architectures, such as PALs where only the AND array is configurable (CHEN, 2006). There are 2^n logic gates in the AND plane, one for each minterm, and m OR gates, where m is the number of outputs on the functions. Each input can be connected to an AND gate directly or through the inverter connected to them. Therefore, the PLA format representing a function implemented in such technology comprises the function on-set of cubes. It is also possible to describe an ISBF with the PLA format by setting outputs as *don't care* by using the sign "-". Figure 2.1 shows the PLA file representing the circuit configuration in Figure 2.2.

```
# number of inputs
.i 3
# number of outputs
.o 1
# input signals name
.ilb x0 x1 x2
# output signals name
.ob y0
# on-set
000 1
001 1
101 1
110 1
111 1
.e
```

Figure 2.1 – PLA format representation of the function described in Table 2.1.

2.2.4 Binary Decision Diagram

Although it is a simple way of representing Boolean functions, the truth table data structure is not scalable since it always uses 2^n bits to store the information. So, the adoption of truth table data structure for Boolean functions representation are not indicated for functions with a high number of input variables. In order to overcome this limitation, Akers proposed the concept of decision diagrams (AKERS, 1978).

A binary decision diagram (BDD) is a rooted, directed acyclic graph used to represent Boolean functions. The graph vertex set V accepts three types of vertices. A terminal vertex v may denote a $TRUE$ or $FALSE$ decision, through the values 1 and 0, respec-

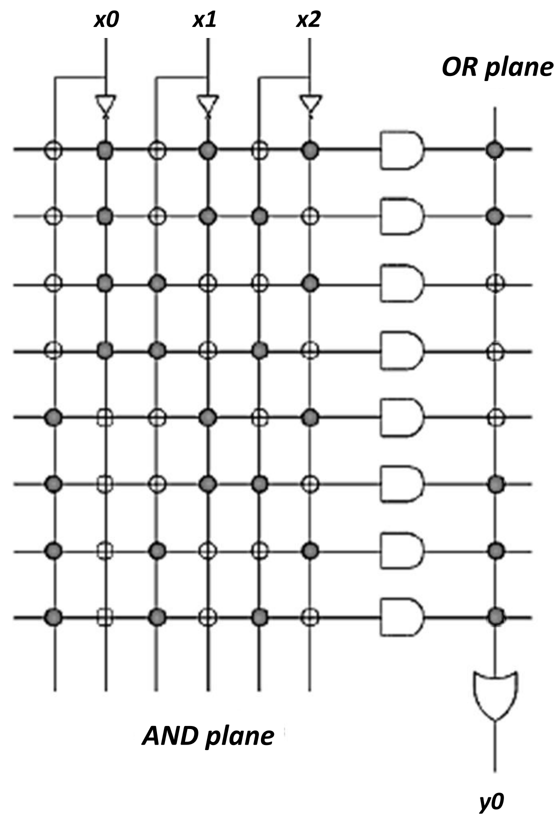


Figure 2.2 – PLA architecture configuration for the function presented in Table 2.1.

tively. On the other hand, a *non-terminal* vertex v has as attribute an input variable x_i and represents a decision node with two children. If x_i = 1, go to $high(v)$, else go to $low(v)$. Finally, there is the function node, which has one incoming edge and no outgoing edges, and denotes the represented function. Figure 2.3 illustrates the BDD representation of the function described in Table 2.1, respecting the variable order $x0 < x1 < x2$. The F node denotes the represented Boolean function. The dashed edges refer to the $low(v)$ nodes.

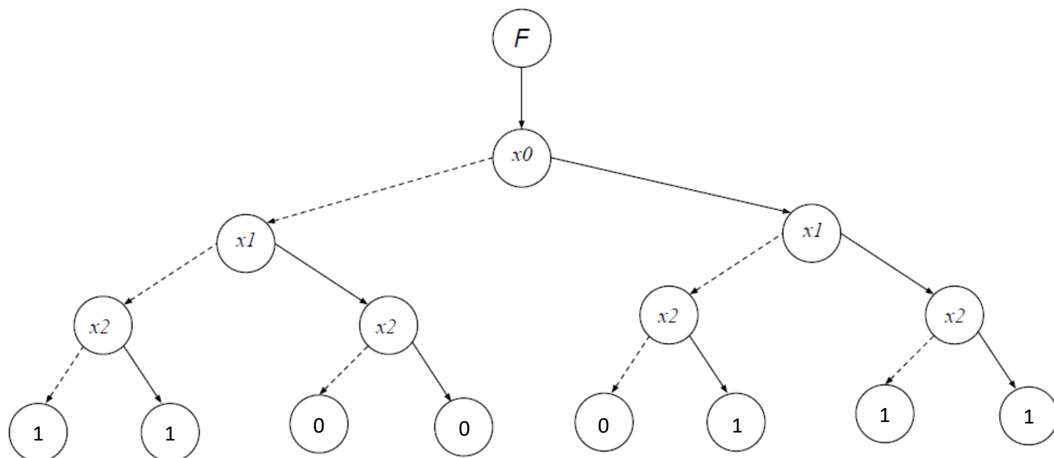


Figure 2.3 – BDD representation of the function described in Table 2.1.

To make the BDD a canonical representation, the concept of reduced and ordered

binary decision diagram (ROBDD) was introduced in (BRYANT, 1986), being a more compact way to represent BDDs. The canonicity is related to a given order of variables in the graph, which directly impacts the ROBDD size (HACHTEL; SOMENZI, 2006). For a given order, it may be the case that it is impossible to derive an ROBDD. For a given variable ordering, *non-terminal* nodes controlled by the same variable and pointing to the same child in both values (xi = 1 and xi = 0) are removed. Moreover, nodes controlled by the same variable and pointing to the same left and right child are merged. A ROBBD for the same function described in Table 2.1 is shown in Figure 2.4.

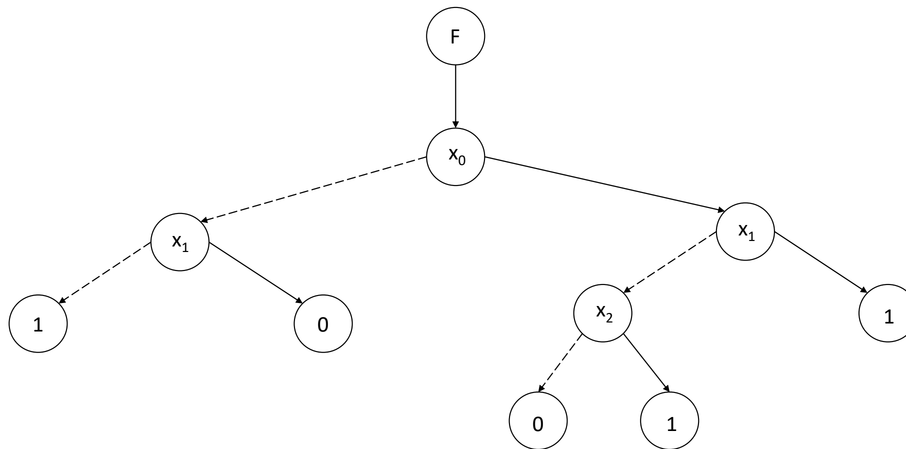
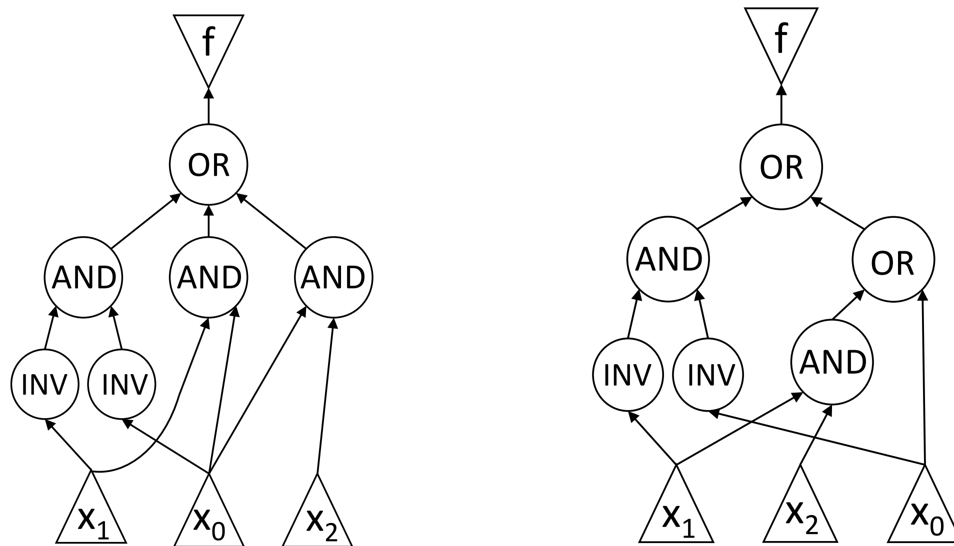


Figure 2.4 – Optimized ROBDD of the function described in Table 2.1.

2.2.5 Boolean Networks

A Boolean network is a directed acyclic graph (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990). A Boolean operation is associated with each node in the graph. A directed arc from node i to node j means that the i-th function output is an input of the j-th function. Some of the graph nodes are designated as inputs and outputs to the network, called the *primary inputs* and *primary outputs*, respectively. Any node that has input and output arcs is an intermediate node. A node can be both an output and an intermediate node. In general, a Boolean network is an implementation or representation of a set of Boolean functions. The Boolean network can be directly derived from the Boolean expression. Moreover, representing a function as a Boolean network shows the importance of reducing the number of literals in the corresponding mathematical expression. Figure 2.5 shows two Boolean networks implementation for the function represented in Table 2.1. Figure 2.5a presents a Boolean network for an SOP expression and Figure 2.5b a

Boolean network for a multilevel expression.



(a) Boolean network for a SOP expression.

(b) Boolean network for a multilevel expression.

Figure 2.5 – Two Boolean network representations of the function described in Table 2.1.

2.2.6 AND-Inverter Graph

The AND-Inverter Graph (AIG) is the most common data structure for performing logic synthesis, even though it was first proposed to perform combinational equivalence checking (KUEHLMANN; KROHM, 1997). An AIG is a directed acyclic graph (DAG) used to represent Boolean functions, and its nodes have zero or two incoming edges. Nodes with zero incoming edges are primary inputs (PI), and nodes with two incoming edges represent the 2-input AND logic operator (AND2). Moreover, nodes can be marked to represent primary outputs (PO). The operators may or may not be inverted. The inversion is represented by complementing the graph edges. An AIG is a particular case of a Boolean network.

In the adoption of AIG structure for logic synthesis algorithms, it is very common to compute k-cuts (PAN; LIN, 1998; MISHCHENKO; CHATTERJEE; BRAYTON, 2006). For a given AIG node n, its cut C is a set of nodes in the network, also known as leaves of the cut, such that every path from a PI to n comprises at least one node in C. An AIG node n can comprise multiple cuts. A cut is said to be k-feasible if it comprises no more than k nodes. Otherwise, the cut is discarded. Usually, the k-cuts are computed into a single pass from PIs to POs, and the computation is performed as follows. If the node

is a PI, it has a trivial cut, i.e., the node itself is the cut. The k-cuts of an AND2 node are given by the Cartesian product between the cut sets in each of its inputs and its trivial cut. Figure 2.6 presents the AIG of the function represented in Table 2.1. The dashed lines represent inverters.

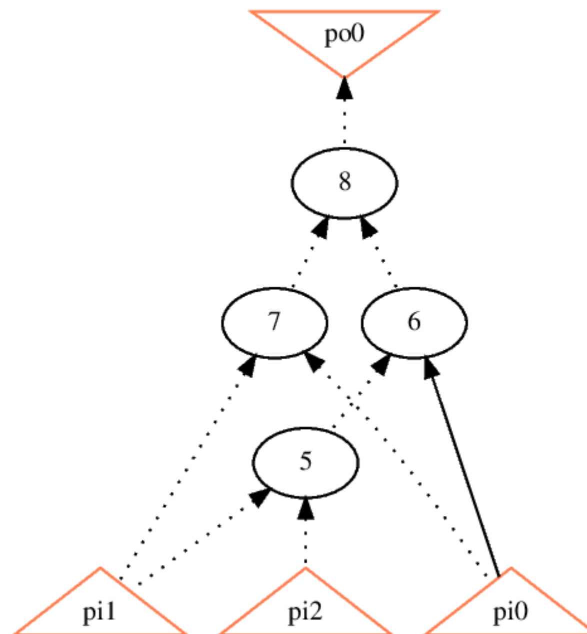


Figure 2.6 – AIG representation of the function described in Table 2.1.

2.3 Two-level Logic Optimization

As two-level expressions have fixed logic depth, the main objective of optimization techniques is to minimize the number of literals and the number of cubes in an SOP context or sums in a POS context. This work focuses on optimizing the number of literals in a cover.

A well-known approach to represent and synthesize a Boolean function in an optimized cover is the Karnaugh map (KARNAUGH, 1953). In a Karnaugh map each line of a truth table is mapped into a cell, and the cell value is the output for the respective line. The cells in the Karnaugh map are ordered using the Gray code, such that the position of neighbor cells differs by exactly one bit, i.e., the Hamming distance equals to one. Figure 2.7 shows the Karnaugh map for the function described in Table 2.1.

The main idea of using a Karnaugh map to optimize a cover is to use rectangles to cover all minterms of ON-SET(f). A rectangle must contain a power of two minterms (i.e., 1, 2, 4, 8, and so on), and each rectangle represents a cube. If a minterm is inside

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	0
1	1	0	1	1

Figure 2.7 – Karnaugh map representation of the function described in Table 2.1.

a rectangle, this minterm is covered by the cube represented by the rectangle. The size of a cube is the number of minterms covered by it. The bigger the cube (rectangle) is, the fewer literals are needed to represent it. In general, the Karnaugh map objective is to select the cover that minimizes the literals count. Figure 2.8 presents the Karnaugh map in Figure 2.7 with all rectangles that cover more than one literal highlighted. Some rectangles only cover one minterm, but as a bigger rectangle covers these cubes, they do not result in the best cover and can be ignored. Note that for this function it is impossible to place a rectangle that covers four minterms.

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	0
1	1	0	1	1

Figure 2.8 – Karnaugh map with all possible cubes highlighted.

Although these cubes are the biggest ones possible, that is not the best cover solution. Note that the rectangles in blue and red only cover minterms that are also covered by other cubes. In this way, by removing one of these cubes do not uncover any minterm and reduce the number of literals of the cover. Figure 2.9 shows an optimal cover for the function after removing the blue rectangle.

In the Karnaugh map approach is easy to visualize the optimization and it can reach an optimal cover, but it is challenging to apply in functions with more than six variables. The Quine-McCluskey algorithm (MCCLUSKEY, 1956), also known as the method of prime implicants, utilizes the same concepts of the Karnaugh map approach without representing the function with the Veitch diagram. This algorithm comprises two phases: (1) finding all prime implicants of the function, and (2) use those prime implicants

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	0
1	1	0	1	1

Figure 2.9 – Karnaugh map containing the SOP $!x_0{*}!x_1 + x_0 * x_1 + x_0 * x_2$

to cover the function.

To find all prime implicants is necessary to combine the previous implicants until any new implicant can be generated. The procedure to find the prime implicants is the following. At first, all minterms are listed and considered as implicants. In the next, it tries to combine every possible pair of neighbor implicants, i.e., two implicants that contain identical literals with only one of them with different polarity. They can be combined, generating a new implicant that does not contain this literal, as in the following example:

$$!x_0 * x_1 * x_2 + x_0 * x_1 * x_2 = (!x_0 + x_0) * x_1 * x_2 = x_1 * x_2.$$

The new implicants are stored for the next iteration, while the original implicants that generated these new implicants are not. When it is not possible to combine neighbor implicants, the remaining implicants are the prime ones. In Table 2.4 is presented the implicants generated in the synthesis of the function described in 2.1. The first column shows the implicants of the function minterms with an identifier in parentheses. The second column presents the new implicants and the identifiers corresponding to the implicants that were combined. There are also the minterms covered by the implicant.

Table 2.4 – Example of finding all prime implicants of the function described in Table 2.1

First Iteration		Second Iteration	
(1)	$!x_0{*}!x_1{*}!x_2$	(1+2)	$!x_0{*}!x_1$
(2)	$!x_0{*}!x_1 * x_2$	(2+3)	$!x_1 * x_2$
(3)	$x_0{*}!x_1 * x_2$	(3+5)	$x_0 * x_2$
(4)	$x_0 * x_1{*}!x_2$	(4+5)	$x_0 * x_1$
(5)	$x_0 * x_1 * x_2$		

Note that in the second iteration, only the implicants $!x_0{*}!x_1$ and $x_0 * x_1$ contain the identical literals. However, as the number of literals with different polarities is more than one, it is impossible to combine them. Hence, the implicants of the second iteration are prime implicants.

The objective of the second phase is to select a set of prime implicants that cover the function and minimizes the number of literals. As essential prime implicants must be in the cover, it inserts them in the solution. The minterms covered by the essential prime implicants are already covered. Then a subset of the remaining prime implicants must be selected to cover the uncovered minterms. The final cover must be irredundant. In Table 2.4, the prime implicants $!x_0*!x_1$ and $x_0 * x_1$ are essential as the minterms $!x_0*!x_1*!x_2$ is only covered by the first and the minterm $x_0 * x_1*!x_2$ by the second. When the essential cubes are inserted in the cover, the only minterm not covered is $x_0*!x_1 * x_2$ and can be covered by both prime implicants $!x_1 * x_2$ and $x_0 * x_2$. As both prime implicants contain two literals, choosing any of them to be added to the cover results in the same number of literals.

The Quine-McCluskey approach focuses on just one function (or output), but the concepts on this approach can be extended to synthesize multiple-output circuits. The main difference between single- and multiple-output functions is the possibility of having output literals on the cubes (implicants). The number of literals in a cube is the sum of the input and output literals. Note that when an output literal is inserted in an implicant, its size doubles. So, the cube size and number of output literals are directly proportional, unlike input literals whose number is inversely proportional to the cube size. When two implicants have identical input literals but different output ones, it is possible to combine them in an implicant containing the input literals and merging the output literals. Moreover, two implicants with identical output literals and input literals with only one input literal with different polarities can be combined. The basic Quine-McCluskey algorithm can be applied considering these modifications, generating the prime implicants by combining them, inserting the essential prime implicants, and selecting the cubed to cover the uncovered minterms. The optimized cover for the multi-output circuit described in Table 2.2 is

$$\{y_0*!x_0*!x_1, y_1*!x_1*!x_2, y_0 * y_1 * x_0 * x_2, y_0 * y_1 * x_0 * x_2\}.$$

It is also possible to adapt the basic Quine-McCluskey approach to synthesize ISBF. In the basic approach, finding prime implicants starts considering the minterms of the ON-SET(f) as implicants. When an ISBF is synthesized, the ON-SET(f) and DC-SET(f) minterms are considered the initial implicants. The remaining approach to find the prime implicant is the same as the basic approach. For the second phase, the main difference is that it only has to cover the ON-SET(f) minterms. In other words, the ON-

SET(f) and DC-SET(f) minterms are used to obtain the prime implicants, but only the minterms of the ON-SET(f) need to be cover. When the objective is to synthesize a multi-output circuit, the use of the DC-SET(f) is done over the multi-output approach. The optimized cover for the multi-output circuit described in Table 2.3 is

$$\{y_0 * !x_1, y_0 * y_1 * x_0\}.$$

2.4 Boolean Satisfiability

Boolean satisfiability (SAT) is the decision problem to determine whether there exists an assignment to the input variables that makes the output of a given Boolean formula evaluates to true. If such an assignment exists, then the formula is *satisfiable*. Otherwise, the formula is *unsatisfiable*. Conventionally, SAT problem instances are represented by a formula in the conjunctive normal form (CNF).

A problem that is closely related to SAT is the model counting, sometimes also referred to as SAT counting or #SAT, which computes the number of satisfiable assignments for a given formula. SAT solvers are tools implementing advanced techniques for deciding whether a given SAT instance is satisfiable or unsatisfiable (EéN; SöRENSSON, 2004). Many practical problems can be modeled using SAT and efficiently solved by modern SAT solvers. This work refers to a tool used to solve the #SAT problem as an #SAT solver.

2.5 Espresso Tool

The ESPRESSO logic minimizer is a framework using heuristic and specific algorithms to optimize two-level circuits efficiently (BRAYTON et al., 1984). ESPRESSO receives as inputs the cube covers of the ON-SET and the DC-SET of an incompletely specified Boolean function. It returns a minimized cover as its output. The objectives of ESPRESSO are to minimize the number of product terms in the cover, the number of literals in the input parts of the cover, and the number of literals in the output parts. The sequence of operations carried out by ESPRESSO is outlined below.

1. Complement: Compute the complement of the ON-SET and the DC-SET, i.e., compute the off-set.

2. Expand: Expand each implicant into a prime and remove covered implicants.
3. Essential Primes: Extract the essential primes and put them in the don't-care set.
4. Irredundant Cover: Find a minimal (optionally minimum) irredundant cover.
5. Reduce: Reduce each implicant to a minimum essential implicant.
6. Iterate 2, 4, and 5 until none of the three metrics have been improved.
7. Lastgasp: Try reduce, expand and irredundant cover one last time using a different strategy. If successful, continue the iteration.
8. Makesparse: Include the essential primes back into the cover and make the its structure as sparse as possible.

3 APPROXIMATE COMPUTING PARADIGM

This chapter presents some concepts and techniques about approximate computing. Initially, error-resilient applications are defined. Then, the approximate computing paradigm is introduced, reviewing some definitions and the different computational levels where approximate computing techniques can be exploited. At the end, the application on the gate level is discussed, presenting existing approaches for building approximate circuits.

3.1 Error-Resilient Application

Error resilience is the characteristic of an application that produces acceptable result besides its execution is incorrect or approximated. Such a characteristic appears in a broad spectrum of applications, such as digital signal processing, image, audio and video processing, graphics, wireless communications, web search, data analytics, RMS (recognition, mining and synthesis), machine learning, sensor-driven application and Internet-of-Things (IoT) (CHIPPA et al., 2013). There are some behaviors that may contribute to the error resilience propriety, which can be roughly divided into five groups:

- Noisy input data: Applications that work over noisy or eventually incorrect input data tend to be robust and generate valid outputs to those erroneous inputs. This behavior is common in applications that interact with real-world data, such as sensor-driven applications.
- Redundant input data: In applications where similar data are processed several times (redundancy), small changes on input data may not impact that result quality. Neural networks is an example of application with this behaviour.
- A golden output does not exist or it is not necessary: The applications with this behavior contain a range of acceptable output values. It is common in applications where the human perception measures the quality of the output information, such as image and audio processing.
- Statistical or probabilistic computation: Applications that use statistical or probabilistic methods tend to attenuate or cancel errors, avoiding eventual erroneous output. The majority of machine learning applications fit in this category.
- Self-healing methods: these applications present an iterative nature of computa-

tions, where errors due to approximations in one iteration can potentially get healed or recovered in subsequent iterations. An example of a self-healing method is neural networks with incremental learning.

3.2 Approximate Computing

Approximate computing is a computational paradigm that allows a system to present imprecise or inexact behavior aiming to simplify the system complexity, making it faster or cheaper. Surveys about approximate computing techniques can be found in (HAN; ORSHANSKY, 2013; MITTAL, 2016; XU; MYTKOWICZ; KIM, 2016). When approximate o computing is applied on error-resilient applications, it is possible to optimize the system without impacting the quality of results.

This paradigm can be applied on several computational levels, from a high level as algorithms and compilers, passing through the architectural level to the circuit level in both logic gates and transistors levels. In general, the approximate approach, the number of allowed errors, and the aimed optimization depend on the approximation level (MITTAL, 2016; XU; MYTKOWICZ; KIM, 2016). The final application characteristics may also impact the approximation approach, but we do not present any application-specific technique in this work. Next, some approaches of approximated computing on each computational level are presented. The approximation on circuit level is shown in Section 3.3.

At algorithm level, it is possible to modify the exact behavior of a program to speed up its execution and reduce the energy consumed. An example of this approximation is using loop perforation (SIDIROGLOU-DOUSKOS et al., 2011), which skips some iterations of a loop, reducing the computation cost. Also exist specific programming languages to approximate computing, being used, for instance, to execute code partition in a probabilistic way (GORDON et al., 2014) or to annotated portions of code that can be (or not) approximated, called critical and not critical regions (SAMPSON et al., 2011; BORNHOLT; MYTKOWICZ; MCKINLEY, 2014).

The approximation at compiler level is made by modifying the program behavior considering the annotations made by the programming language or by selecting where the approximation can be made (XU; MYTKOWICZ; KIM, 2016). The approximations in this level can be modifications on the machine code (MISAILOVIC et al., 2010) or pointing code parts to be executed on approximate hardware (MISAILOVIC et al., 2014),

for instance.

The approximation can be made in different ways at architectural level, such as modifying the exact behavior of components, like processors, storage and graphic processing unit (GPU), and using neural processing unit (NPU) to perform approximate executions. The change of behavior in almost all components can be made by reducing the voltage supply. Change the voltage supply reduces energy consumption in exchange for errors due to timing in most cases (MITTAL, 2016; XU; MYTKOWICZ; KIM, 2016).

The approximation in a processor can be made by creating operations at the instruction set architecture (ISA) that compute approximate operations, which can use approximate hardware or the original approximate operation (VENKATARAMANI et al., 2013; ESMAEILZADEH et al., 2012b). A processor can be provided with both precise and approximate data paths or even whole approximate cores (ESMAEILZADEH et al., 2012b; KARPUZCU; AKTURK; KIM, 2014). Hence, it is possible to interchange exact and approximate operations at the same processor, being also possible to have multiple paths with approximation strategies with different error constraint.

Some examples of approximation on storage elements are observed by changing the DRAM refresh rate (CHO et al., 2014), changing error correction algorithms in SSD memories (XU; HUANG, 2015), and modifying allocation algorithms in non volatile memory (NVM) by using blocks with exhausted error-correction resources to store error-resilient data (SAMPSON et al., 2013).

An NPU is a specialized circuit applied to optimize the execution of machine learning algorithms. It can be used to execute an approximate version of a set of CPU operations (ESMAEILZADEH et al., 2012a; AMANT et al., 2014). This approximation is made by training a neural network that mimics the exact behavior of the set of CPU operations and executing on the NPU. The NPU execution tends to be faster and consumes less energy than the CPU but presents an error rate equals to the neural network accuracy.

3.3 Approximate Circuits

The main focus of the present work is to use approximate computing at circuit level, the called *approximated circuits*. The approximation at circuit level includes approximation at gate level and transistor level. It consists of deriving a circuit with a Boolean behavior different from the original specification. When the approximation is made in a good way, it is possible to obtain optimizations in area, performance and

energy efficiency compared to an implementation Boolean equivalent to the specification (SCARABOTTOLO et al., 2020; HAN; ORSHANSKY, 2013; XU; MYTKOWICZ; KIM, 2016; MITTAL, 2016).

There are two main approaches to approximate a circuit: by modifying the voltage supply, similar to the architectural level approaches, and by modifying the Boolean function implemented by the circuit, aiming at a function that can be represented in a more optimized way. The latter one can be made by approximating manually or automatically the circuit design.

The first efforts in circuit approximation were the result of manually approximating the target design. Such a handcrafted approximation demands the circuit designer to have deep knowledge about the circuit behavior to make approximations that optimize it but do not degrade the output quality.

The main application of manual approximation is arithmetic designs. There are numerous efforts to approximate adders and some works that aim at multipliers and dividers. Reviews comparing different approximation approaches of adder and multiplier designs can be seen in (JIANG; HAN; LOMBARDI, 2015) and in (JIANG et al., 2016), respectively. In (MRAZEK et al., 2017), the authors present a library with different approximate adders and multipliers.

Many approximation schemes have been proposed by reducing the critical path and hardware complexity with respect to accurate adder. There are two main strategies to approximate an adder circuitry: by modifying the carry chain structure, mainly to reduce the propagation delay or the cost of carry prediction calculation, and by approximating the full-adder (FA) design as it represents the basic block for adder designs (JIANG; HAN; LOMBARDI, 2015).

In (MOHAPATRA et al., 2011), the authors approximate an adder circuitry by segmenting it to reduce the carry chain delay. An N-bits adder is split into K sub-adder of N/K bits, reducing in K times the adder delay. This work also proposes the use of multiplexers (MUX) between the sub-adders to select a fixed carry-in or the carry-out from the last sub-adder, and so controlling the error introduced by the segmentation. Moreover, the approximation of FA behavior in the least significant bits (LSBs) of the adder structure to perform an OR operation is proposed in (MAHDIANI et al., 2010). The FA approximation procedure has been also applied to create approximate floating-point adder (LIU et al., 2014).

Multiplier designs can employ 2x2 elementary multiply modules to generate par-

tial products. An adder tree is used to add the partial products and obtain the product result for multiply operations (REHMAN et al., 2016). In (KULKARNI; GUPTA; ERCEGOVAC, 2011), it is presented the design of an inexact 2x2 multiplier that represents the multiplication of $3_{10} * 3_{10}$ with 7_{10}, instead of 9_{10}. In other words, 11_2*11_2 is represented by 111_2 instead of 1001_2, which uses three bits instead of four. For the remaining 15 input combinations, the multiplier provides the correct output. Thus, while still providing a correct output for 15 out of 16 input combinations, their inexact multiplier reduces the area by half compared to the exact version, leading to a shorter and faster critical path. In (BHARDWAJ; MANE; HENKEL, 2014), the authors present an approximate multiplier design by approximating the x-LSBs related partial products of the adders to optimize the target design.

As done in approximate multiplier designs, approximate dividers can also be derived using approximate adders (PARHAMI, 2009) and approximate elementary divider modules (CHEN et al., 2016). In (SAADAT; JAVAID; PARAMESWARAN, 2019), it is proposed a modification on the approximate log-based divider technique, a classical technique to create approximate dividers, to reduce the amount of inserted error, and derive optimized integer and floating-point approximate dividers.

3.4 Approximate Logic Synthesis

Section 3.3 presented the basic concepts about approximate circuits and the existing works that propose approximations on arithmetic designs by manually modifying the circuit structure. The main drawback of this approach is that it is necessary design expertise knowledge about the circuit behavior to approximate it, turning it into a design-specific technique. On the other hand, the automatic circuit approximation approach aims to overcome these drawbacks.

The synthesis of approximate circuits consists of automatically modifying a general logic behavior without knowing the implementation details. It is done by systematically modifying the behavior without exceeding a given error threshold. Such a systematic manipulation is usually similar to problems tackled in logic synthesis tools. In that way, the automatic generation of approximate circuits is usually called *approximate logic synthesis* (ALS) (SCARABOTTOLO et al., 2020).

In general, ALS methods apply modifications to the logic behavior iteratively until it is impossible to modify the circuit without exceeding the error threshold. Hence, besides

the systematic approximation approach, it is needed to apply techniques to calculate the error. The error calculation method is usually applied over each possible modification during the approximation process and depends on the error metric applied. The most adopted error metrics in ALS and the different approaches to calculate the introduced error during approximation task are presented in Section 3.5.

In literature, there are ALS efforts to approximate combinational circuits for both two-level and multilevel representations as well as for sequential circuits. The rest of this section focus on ALS works for combinational multilevel circuits at first and then sequential circuits. The works that focus on the approximation of two-level combinational blocks are presented in Section 3.6.

In (SHIN; GUPTA, 2011), it is proposed the first ALS method focusing on multilevel circuits. This work applies hardware testing techniques to enumerate the modifications performed on the circuit, which can be seen as possible errors in the test of hardware context. It is done by selecting the changes that lead to optimizations on the circuit with little impact on the resulting output quality.

In (VENKATARAMANI et al., 2012), two significant contributions can be found: it formulates the ALS problem and it is the first approach that applies logic synthesis techniques to approximate the circuit. This work performs the approximation process by exploiting some concepts and techniques of optimizations through *don't cares*, a recurrent problem on conventional logic synthesis. In (MIAO; GERSTLAUER; ORSHANSKY, 2014), the authors propose an ALS approach that also uses these concepts and techniques.

An ALS approach that identifies pairs of internal signals, i.e., that assume the same value with high probability and substitute one for the other, is presented in (VENKATARAMANI; ROY; RAGHUNATHAN, 2013). While these substitutions introduce functional approximations, they may result in some logic to be eliminated from the circuit. A modification on this approach that uses a more precise error calculation method, leading to better circuit optimization for the same error threshold, is presented in (SU; WU; QIAN, 2018).

In (SOEKEN et al., 2016) and in (WENDLER; KESZOCZE, 2020), the authors propose approaches to approximate circuits represented by BDDs. These works detail procedures to modify the BDD structure as well as to estimate the inserted error over BDDs.

An ALS approach over Boolean networks with approximation being made in a two-level representation is proposed in (WU; QIAN, 2016). For a logic behavior rep-

resented by a Boolean network, subcircuits in the networks are selected and converted into a two-level Boolean expression. The obtained expression is approximated by removing literals and then is returned to a multilevel representation, substituting the original subcircuit. An extension of this work is presented in (WU; QIAN, 2020).

(WU et al., 2017) proposes an ALS approach focusing on FPGA mapped circuits. During the mapping to FPGA, subcircuits of the original circuit are mapped on LUTs, which have a limited number of inputs. In that way, the main idea is to remove input signals in subcircuits that lead to multiple LUTs to reduce the number of LUTs needed to map the circuit.

In (LIU; ZHANG, 2017), it is presented a framework that approximate the circuit by using stochastic heuristic and by apply statistical test during the approximation process in order to obtain a solution that respects the error threshold with a high degree of confidence. It represents the first work to take into account different input distributions at the input during the approximation task.

In (YAO et al., 2017), the authors present an ALS approach to approximate a Boolean function aiming to turn it on a maximally disjoint bi-decomposable Boolean function. When a Boolean function represents a maximally disjoint bi-decomposable, the circuit may have a more optimized implementation. In that work, the objective is to modify the function characteristic instead of modifying the circuit representation, as done in other works.

In (CHANDRASEKHARANA et al., 2016) and in (ZHOU et al., 2018), the authors propose ALS methods that focus on critical paths of AIG, aiming to reduce the circuit logic depth. Both approaches select subgraphs on the AIG critical path to replace for a constant value, so reducing the logic depth and the number of nodes. The difference between these works is that the subgraph selection is optimized on the latter one.

An ALS approach proposed to optimize AIG by focusing on the number of nodes is presented in (MENG; QIAN; MISHCHENKO, 2020). The optimizations in this work are related to a set of input simulation vectors and comprise two phases. The first one identifies signal pairs that assume the same values over the input simulation vectors and substitute one for the other. The second phase optimizes a subgraph comprising the substituted signal by using the Espresso tool. Such an optimization is done over a truth table inducted by observing the inputs and the respective output on the subgraph while simulating it. If an input combination does not occur in the subgraph, it is considered a don't care.

When the objective is to approximate sequential circuits, there are a limited number of works in the literature. The main reason for this tiny amount of efforts is the difficulty of calculating and controlling the error introduced during approximation because the circuit approximation can result in an error after multiple clock cycles or only in a specific circuit state. In (RANJAN et al., 2014), it is proposed an ALS approach for sequential circuits that applies combinational multilevel ALS approaches, as proposed by (VENKATARAMANI et al., 2012), and voltage scaling approximate techniques to approximate more proper circuit regions. This work also proposes a specific method to calculate the error introduced over sequential circuits. In (CHANDRASEKHARAN et al., 2016), it is proposed the use of approximate combinational circuits in a sequential circuit, and also presents an error calculation technique.

3.5 Error Modeling

Independent of the ALS approach, whether it focuses on two-level or multilevel representation or which Boolean structure is adopted to represent the circuit, it is necessary to define a way to quantify (and control) the amount of error inserted during the approximation procedure. This section presents the error metrics used to quantified the error and the main methods to calculate it during the circuit approximation.

3.5.1 Error Metrics

Many error metrics have been proposed in the literature to control the error inserted in approximate circuits. The error metrics to approximate circuits can be divided into general purpose metrics, which are related to the error occurrence, and arithmetic metrics, which are related to the error magnitude (VASICEK, 2019). This subsection presents the seven most common error metrics adopted (MRAZEK et al., 2017; FROEHLICH; GROßE; DRECHSLER, 2019a). For the expressions presented in the next, n represents the number of circuit inputs, $\mathbb{B}^n$ represents all possible input combinations, $f(x)$ and $\hat{f}(x)$ represent the output value in the original and approximate circuit, respectively, for the input combination x, with $f_i(x)$ and $\hat{f}_i(x)$ representing the i-th output bit. The first two error metrics are for general propose and the five remaining for arithmetic propose:

- Worst Bit Flip (WBF): Also referred to as the worst Hamming distance (WHD). It represents the greatest Hamming distance between the original and the approximate output. It is used to verify which is the worst-case in the number of erroneous output bits. WBF is calculated by

$$WBF = \max_{\forall x \in \mathbb{B}^n} \sum_{i=0}^{n-1} f_i(x) \oplus \hat{f}_i(x).$$

- Error Rate (ER): Represents the frequency or the probability of observing one or more erroneous output bits. ER is the most common error metric for circuit approximation, widely used in general and arithmetic circuits. ER is calculated by

$$ER = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} f(x) \neq \hat{f}(x).$$

- Worst-Case Error (WCE): It consists of the greatest absolute difference in magnitude between the original and the approximate circuit output. It is the most common metric for arithmetic circuits. WCE is calculated by

$$WCE = \max_{\forall x \in \mathbb{B}^n} |f(x) - \hat{f}(x)|.$$

- Mean Absolute Error (MAE): It consists of the average absolute difference in magnitude between the original and the approximate circuit outputs regarding all input combinations. MAE is calculated by

$$MAE = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} |f(x) - \hat{f}(x)|.$$

- Worst-Case Relative Error (WCRE): It consists of the greatest relative difference in magnitude between the original and the approximate circuit outputs. Relative error metrics are interesting because it is a percentual value, making it possible to analyze circuits with a different number of inputs without defining a fixed magnitude value. WCRE is calculated by

$$WCRE = \max_{\forall x \in \mathbb{B}^n} \frac{|f(x) - \hat{f}(x)|}{\max(1, f(x))},$$

with the division by $\max(1, f(x))$ being used to prevent a division error when the original output is zero.

- Mean Relative Error (MRE): It consists of the average relative difference in magnitude between the original and the approximate circuit outputs over all input combinations. MRE is calculated by

$$MRE = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \frac{|f(x) - \hat{f}(x)|}{\max(1, f(x))}.$$

- Mean Squared Error (MSE): It consists of the average quadratic difference in magnitude between the original and the approximate circuit outputs. The average quadratic error is used to prevent large errors from occurring, as each error is squared. MSE is calculated by

$$MSE = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} (f(x) - \hat{f}(x))^2.$$

3.5.2 Error Calculation Methods

At the end of Section 3.4, it is mentioned that the main limitation for sequential circuits is the error calculation. In combinational circuits, the error calculation is less complex than for sequential ones because the error does not depend on clock cycles. Nevertheless, the error calculation during the approximation of combinational blocks is the basis and the most computationally expensive phase in ALS procedure.

Some ALS methods calculate in an accurate way the error by using formal procedure, whereas others calculate an imprecise error value. In this work, the approaches for precise error calculation are called error calculation methods, while the imprecise error calculation ones are denominated error estimation methods. The error calculation methods are implemented using formal procedures, like Boolean satisfiability (SAT), symbolic computer algebra (SCA), and binary decision diagram (BDD), or even all possible inputs combination, such as truth tables and exhaustive simulations. The error estimation methods are usually implemented by applying Monte Carlo simulation, although there are other works that propose specific approaches.

The choice of which error method is the most appropriate has to take into account a trade-off between quality of results and computational complexity. A precise error calculation tends to result in a better circuit approximation because it leads to a better choice of which approximation is applied during the ALS execution (SU; WU; QIAN, 2018). Formal approaches usually present a high computational cost, and their execution

over each possible modification on the circuit may result in high computational overhead on the ALS method.

When error estimation methods are adopted during the circuit approximation, there is no guarantee that the final solution respects the error constraint. In this case, it is interesting to apply an error calculation method to formally verify if the error constraint is satisfied (VASICEK, 2019). This section presents the main strategies to calculate and estimate the errors during the circuit approximation, associating to the ALS approach where it is applied. Some ALS methods apply specific strategies to estimate the error, and that will not be discussed in this section (SHIN; GUPTA, 2011; VENKATARAMANI et al., 2012; VENKATARAMANI; ROY; RAGHUNATHAN, 2013; MIAO; GERSTLAUER; ORSHANSKY, 2014; WU; QIAN, 2016; WU et al., 2017; YAO et al., 2017).

Techniques based on Monte Carlo simulation are the most common error estimation methods in ALS approaches (LIU; ZHANG, 2017; SU; WU; QIAN, 2018; ZHOU et al., 2018; MENG; QIAN; MISHCHENKO, 2020). The techniques that use Monte Carlo simulation consist of simulating the circuit with random stimuli that do not cover all possible input combinations. The higher is the number of simulations, the greater is the confidence degree that the behavior observed by simulation is equal to the circuit behavior. In other words, using more simulations leads to a more precise error estimation. As expected, more simulations increase the computational overhead. This freedom to trade-off precision and scalability is the main reason for the popularity of Monte Carlo based techniques. For instance, in (SU; WU; QIAN, 2018), the authors propose an approach based on Monte Carlo simulation that focuses on DAG Boolean structures. This approach only simulates DAG regions modified by an approximation, so reducing the simulation overhead and increasing the number of interactions.

A *miter* is a structure frequently used to verify the Boolean equivalency of two circuits (BRAND, 1993). It consists of two circuit implementations connected to the same inputs and with exclusive-OR (XOR) gates at the outputs, pair-by-pair. The XOR outputs are connected to an OR gate. If any XOR output is a true value for a given input combination, the miter output is true, and so not equivalent. In opposite, if any XOR output results in a false value for all input combinations, the circuits are logically equivalent. A recurrent concept in many error calculation methods is the approximate version of a miter, called *approximate miter*. An approximate miter is similar to the miter, with two circuits connected at the same inputs, but it does not connect the outputs of the circuits through an XOR gate. The outputs of the circuits are connected to specific

circuits, which only results in a true value when a given error constraint is exceeded. In Figure 3.1a is shown the structure of a miter.

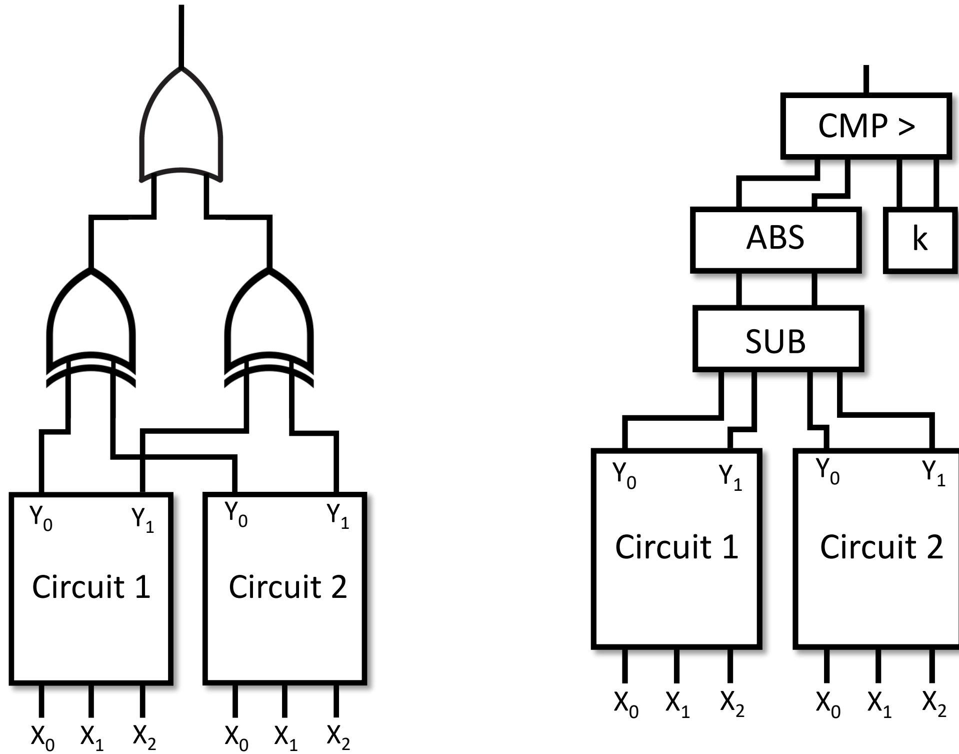
Multiple ALS approaches that adopt SAT-based error calculation methods use approximate miter to build the input CNF for the SAT-solver (CHANDRASEKHARANA et al., 2016; RANJAN et al., 2014; CHANDRASEKHARAN et al., 2016; ČEšKA et al., 2017). In general, the circuit of the approximated miter is converted into a CNF by using the Tseitin transformation (TSEITIN, 1983). In that way, if the SAT-solver finds an input combination where the generated CNF is satisfied, the error between the two circuits in the approximate miter exceeds the error threshold. If the CNF is not satisfiable, the error threshold is not exceeded. The use of formal techniques over an approximate miter is also called relaxed equivalence checking (VASICEK, 2017). In the next, it is presented some examples of calculating ER and WCE metrics using SAT and approximate miter.

For the ER constraint, it is used the same miter as in Boolean equivalence checking, as shown in Figure 3.1a, but it is considered the #SAT problem. In that way, the #SAT-solver does not return all input combinations that result in different outputs. The ER is the number of input combinations divided by the number of possible input combinations.

For the WCE constraint, the outputs are connected to a subtractor, obtaining the difference in magnitude between the circuits. The subtractor output is connected to a circuit to calculate the absolute value. The approximate miter output consists of a comparator that results in a true value if the absolute difference is greater than a fixed value k, where k is the WCE threshold. This structure is illustrated in Figure 3.1b. The SAT-solver is applied over the obtained CNF. If the CNF is satisfiable, the circuits exceed the defined WCE.

When BDDs are used to represent the circuit, there are two ways to calculate the error: using an approximate miter (VASICEK; SEKANINA, 2016), and traversing the BDD (SOEKEN et al., 2016; WENDLER; KESZOCZE, 2020). In the first case, the approximate miter is constructed through the BDD representation, connecting the original and the approximate BDDs. Then an SAT-solver or #SAT-solver is applied over the BDD. Most BDD packages support SAT operation directly over the DAG, being unnecessary to obtain a CNF. The second case modifies the BDD and uses operations over this DAG to calculate the error. As in approximate miter, the modification over the BDD is specific for each error metric.

Symbolic computer algebra (SCA) is a computational area that addresses symbol-



(a) The miter structure used to check if two circuits are logically equivalent. It can be used to calculate the ER between two circuits.

(b) An approximate miter used to check if two circuits do not exceed a WCE threshold k.

Figure 3.1 – Example of two miter structures.

ically by manipulating mathematical equations and expressions without defining values for the variables. SCA is applied in techniques of Boolean verification by dividing the polynomial representation of circuit specification and the corresponding implementation. When the division remainder is zero, the circuit specification and implementation are equivalent. In the context of approximate circuits, the remainder that is not equal to zero represents the difference between the circuits, and can be analyzed considering a given error metric (FROEHLICH; GROßE; DRECHSLER, 2018). In (FROEHLICH; GROßE; DRECHSLER, 2019a), it is presented a technique based on SCA to calculate all error metrics presented in Subsection 3.5.1 by representing the remainder with algebraic decision diagrams (ADD) and traversing it to calculates the different error metrics.

The previously presented error estimation and calculation methods are used on multilevel ALS approaches. Although they can be used in 2L-ALS methods, it is commonly not done because the two-level approaches usually store the function truth table during the circuit approximation (SHIN; GUPTA, 2010; MIAO; GERSTLAUER; ORSHANSKY, 2013; ZOU; QIAN; HAN, 2015; SU et al., 2020). In that way, the error calculation is done by comparing the truth tables of the original and the approximated circuit. As the truth table contains the output results for all possible input combinations,

it is equivalent to an exhaustive simulation.

3.6 Two-Level Approximate Logic Synthesis

Currently, there are four works that propose approaches to approximate two-level circuits. In general, the objective of these works is to obtain an approximate SOP expression (cover) with a minimized number of literals that respect a giver error constraint concerning the original SOP expression (cover). Among these works, one considers WCE and ER metrics together as the error constraint (MIAO; GERSTLAUER; ORSHANSKY, 2013), and the other three ones only consider the ER metric (SHIN; GUPTA, 2010; ZOU; QIAN; HAN, 2015; SU et al., 2020). In this section, we present details on these four 2L-ALS approaches, starting with the WCE and ER approach and then presenting the fundamental behavior of the three ER approaches.

3.6.1 ER and WCE Bounded Methods

The approach presented in (MIAO; GERSTLAUER; ORSHANSKY, 2013) performs the approximation in two phases. The first phase approximates the circuit focusing on the WCE constraint, whereas the second phase undoes approximations realized on the first until the ER constraint is satisfied.

The approximation strategy in the first phase is based on mapping the approximation with the WCE constraint in a Boolean relation problem. The Boolean relation problem is a widely studied problem on synthesis logic. It consists of a two-level circuit where the input combinations can lead to one or more output values. The ISBFs are a specific case of a Boolean relation as the output can assume both logic values. In Table 3.1 is presented an example of a Boolean relation. Note that the possible output for the first input combination can be represented by using the ISBF notation (0-), but it cannot be done for the outputs of the third and fourth input combinations.

Table 3.1 – Example of Boolean relation.

x_0, x_1	Possible outputs (y_0, y_1)
00	{00, 01}
01	{11}
10	{00, 11}
11	{01, 10}

This work shows that the problem of approximating a cover with WCE by applying the error constraint is isomorphic to the Boolean relation problem. When the WCE is taken into account as the error metric, the magnitude difference between the approximate circuit and the original outputs must be less or equal to a given magnitude value M for any input combination. In other words, for each input combination, the approximate circuit output must be into the range (O-M, O+M), where O is the output in the original circuit for the same input combination.

The mapping of the approximation problem into the Boolean relation is done by setting the output value of each input combination to all values into the range (O-M, O+M). In Table 3.2 is shown how this mapping is done, considering a 2-bit adder and the WCE threshold M of 1. After obtaining the Boolean relation, a solver is applied to obtain an optimized cover. Boolean relation solvers aim to select the outputs for each input combination that minimize the literal count in that cover. Several solvers have been proposed as exact approaches based on the Quine-McCluskey algorithm (BRAYTON; SOMENZI, 1989), and heuristics with more scalable behavior (BANERES; CORTADELLA; KISHINEVSKY, 2019).

Table 3.2 – Example of converting a 2-bit adder with WCE threshold of 1 in a Boolean relation.

a, b	Original outputs (c, s)	Possible outputs (c, s)
00	00	{00,01}
01	01	{00,01,10}
10	10	{00,01,10}
11	11	{01,10,11}

As Boolean relation does not have any restriction with respect to the ER. The second phase consists of undoing approximations made in the first phase until the approximate circuit satisfies the ER threshold. It is done by uncovering minterms in which the output value changed from 0 to 1. Similarly, it has to cover the minterms that have changed their output value from 1 to 0. When a correction is done, all outputs of a given input combination must be corrected at the same time to prevent changing the WCE. Thus, this work proposes a greedy procedure to select the input combinations that the correction inserts few literals until the ER constraint is satisfied.

The error calculation used in this work compares the original truth table with the approximate one after the first phase to obtain the ER. The WCE is controlled by constructing the Boolean relation and by correcting all outputs simultaneously in the second phase. Hence, the WCE does not need to be further verified. The error rate is reduced in the second phase based on how many input combinations are corrected, until the threshold

is reached, and there is no explicit error calculation method execution.

3.6.2 ER Bounded Methods

This subsection presents the 2L-ALS approaches that consider ER as the error constraint. For a given cover $\mathcal{C}$ and an ER threshold TE, the problem addressed by these works consists of obtaining an approximate cover $\mathcal{C}'$ that minimizes the number of literals still respecting the ER threshold. In general, the ER threshold TE is a percentage value, but the works presented here approximate the cover using a fixed number of errors (NoE) value as the error threshold. Considering an NoE threshold e, the relation between e and TE are given by $e = TE * 2^n$, where n is the number of inputs of the function. When the approximation is done over an single-output function, it can be represented by an SOP expression. The examples in this subsection will consider the cover presented in the Karnaugh map shown in Figure 3.2 as the original cover $\mathcal{C}$.

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	0
1	1	0	1	1

Figure 3.2 – Karnaugh map containing the cover used on examples. The SOP represented by this Karnaugh map is $!x_0 * !x_1 + x_0 * x_1 + x_0 * x_2$

There are two main techniques to perform the approximation by considering the ER metric: the insertion and removal of cover cubes (SHIN; GUPTA, 2010). The first one inserts new cubes that cannot be inserted without adding errors. If the cube insertion is done cleverly, it implies removing cubes and optimizing the cover. In general, this technique is performed by complementing the output value of minterms from 0 to 1, and verifying which new cubes can be included after such a complement. In Figure 3.3 is shown one possible approximation on $\mathcal{C}$ with an NoE threshold of 1. In Figure 3.3a, the minterm $!x_0 * x_1 * !x_2$, in red, is chosen to have its output complemented from 0 to 1. When this is done, it is possible to insert some new cubes, and the cubes x_1 and $!x_0$ are added in the cover. These cube insertions makes that the cubes $x_0 * x_1$, $x_0 * x_2$ and $!x_0 * !x_1$ becomes redundant, so allowing their removal. These insertions and removals of cubes create an approximate cover $\mathcal{C}'$, shown in Figure 3.3b. Note that the number of literals is reduced

from six to two with one inserted error.

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	1
1	1	0	1	1

(a) The minterm $!x_0 * x_1 * !x_2$ is chosen to have its output complemented from 0 to 1, inserting one error

x_2 \ x_0x_1	00	01	11	10
0	1	0	1	1
1	1	0	1	1

(b) Approximate cover $\mathcal{C}'$ resulting from the application of the cube insertion technique. The SOP of $\mathcal{C}'$ is $x_0 + !x_1$

Figure 3.3 – Example of an approximation done by a cube insertion technique.

The second technique removes cubes directly from $\mathcal{C}$ without previously inserting new cubes. The number of errors added by removing a cube equals the number of minterms that only the removed cube covers on the cover. This technique can be performed by complementing the output value of minterms from 1 to 0, and verifying which new cubes can be removed from $\mathcal{C}$. In Figure 3.4 is shown an approximation on the $\mathcal{C}$ with an NoE threshold of 1. In Figure 3.4a, the minterm $!x_0 * x_1 * !x_2$ is chosen to have its output complemented from 1 to 0. When this is done, the cube $x_0 * x_1$ can be removed from $\mathcal{C}$. After the cube removal, the resulting cover is the approximate cover $\mathcal{C}'$, shown in the Figure 3.4b. In that case, the number of literals is reduced from six to four with one inserted error.

x_2 \ x_0x_1	00	01	11	10
0	1	0	0	0
1	1	0	1	1

(a) The minterm $x_0 * x_1 * !x_2$ is chosen to have its output complemented from 1 to 0, inserting one error

x_2 \ x_0x_1	00	01	11	10
0	1	0	0	0
1	1	0	1	1

(b) Approximate cover $\mathcal{C}'$ resulting from the application of the cube removal technique. The SOP of $\mathcal{C}'$ is $!x_0 * !x_1 + x_0 * x_2$

Figure 3.4 – Example of an approximation done by a cube removal technique.

In (SHIN; GUPTA, 2010), it is presented an experiment showing that the cube insertion technique results in an optimization equivalent or better in comparison to the cube removal technique. In the examples show in Figure 3.3 and in Figure 3.4 confirms the advantage of the cube insertion technique. Based on this experiment, the subsequent works focused on the cube insertion technique (ZOU; QIAN; HAN, 2015; SU et al., 2020).

Based on the approaches presented in the works addressed by this subsection, some steps are recurrent on cube insertion techniques: enumeration of the cube to be inserted without exceeding the error threshold, the combination of these cubes to utilize all allowed errors, the identification of cubes to be removed after the cube insertion, and the calculation of literal reduction after the cube insertions and removals.

The straightforward way to enumerate the cubes to be inserted into cover is to consider all possible cubes for a function and verify which cubes do not exceed the error threshold. This enumeration can be done by exhaustively traversing a Hasse diagram, which consists of a DAG where the vertices are the cubes of a function with n-inputs and m-outputs. The vertices are placed in layers based on their size, with the largest cube on the top and the smallest cubes at the bottom. The top cube is the cube that covers all minterms, and the bottom cubes are the minterms. Exist an edge between two vertices if the cubes differ in only one literal. In Figure 3.5 in shown a Hasse diagram for a 2-inputs 2-outputs function. Note that for a cube in a given layer, the cubes connected to it on the below layer contain one more input literal or one less output literal. Figure 3.6 shows the Hasse diagram for a function with thee inputs and one output, where the cubes in $\mathcal{C}$ and the covered minterms are highlighted in red and orange, and the cubes in green represent the cubes enumerated by an exhaustive search. This technique guarantees that the best possible solution can be obtained as it tests every possible cube. As this approach is exhaustive, it can be computationally costly as for a function with n-inputs and m-outputs, being the number of cubes proportional to $3^n * 2^m$. The approaches presented in (ZOU; QIAN; HAN, 2015) and in (SU et al., 2020) apply Hasse diagram traversing techniques in the cube enumerate phase.

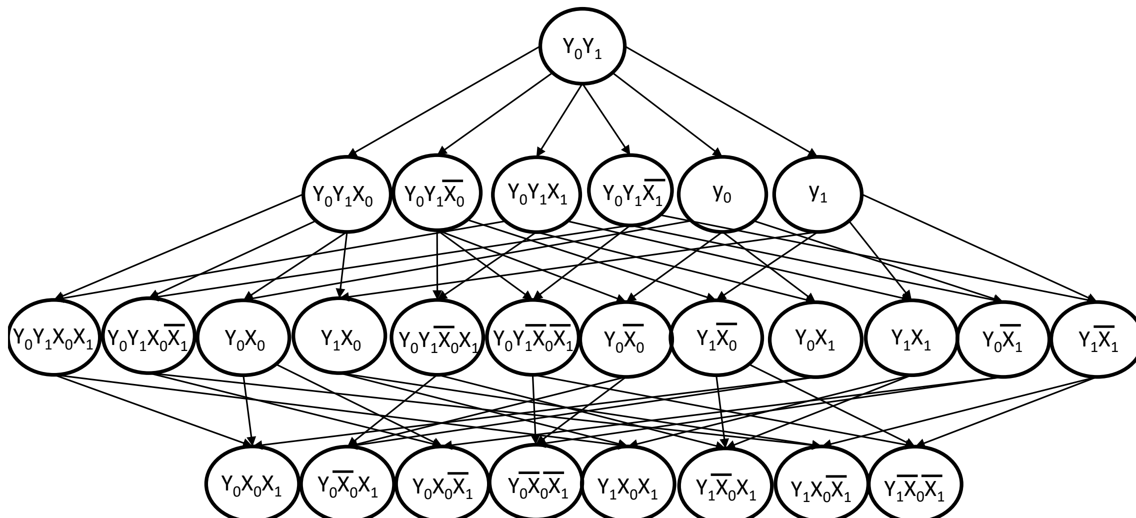


Figure 3.5 – Hasse diagram for a function with 2 inputs and 2 outputs.

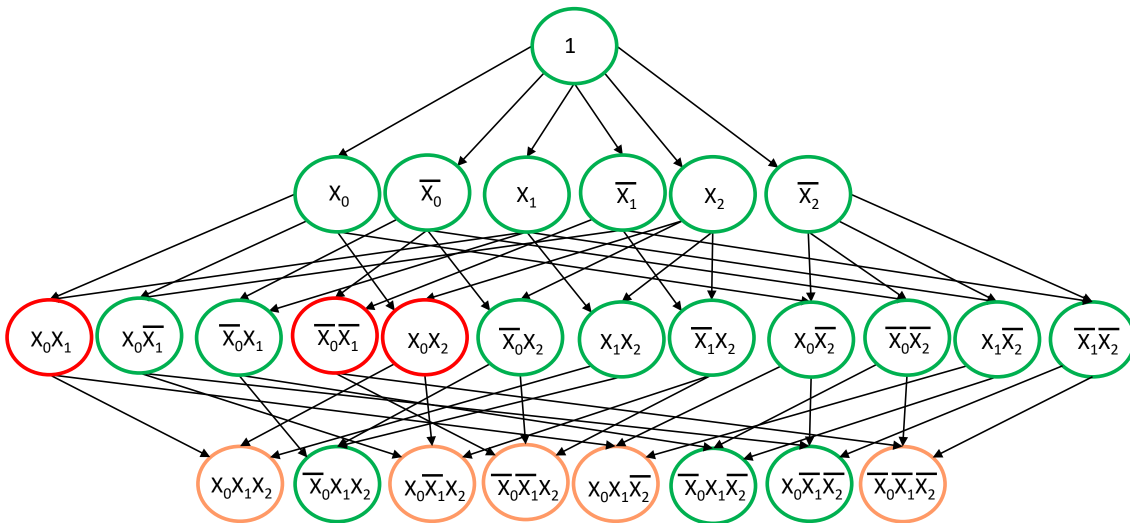


Figure 3.6 – Hasse diagram of 3-inputs 1-output function with the cubes that can be enumerated by the exhaustive technique when using $\mathcal{C}$ as the original function highlighted in green.

Another way to enumerate the cube to be inserted is by expanding the cubes on the cover. Expand a cube consists of removing input literals or adding output literals to increase the cube size, i.e., to cover more minterms. An enumeration based on the cube expansion technique is used in (SHIN; GUPTA, 2010). In a Hasse diagram, expanding a cover cube is equivalent to obtain all cubes in paths between the cube to be expanded and the cube on the top. It is possible to restrict the number of expanded cubes by limiting the number of literals modified in a given cube. For instance, when it is possible only to modify one literal of a cube, the cubes connected with this cube in the one layer above are obtained. Figure 3.7 shows the Hasse diagram of a 3-inputs 1-output function with the cubes that an expansion technique can enumerate over the cubes in $\mathcal{C}$. The cubes in blue in the second layer were obtained by expanding the cubes in $\mathcal{C}$, which are in red. The cube on the top is obtained by expanding the cubes in the second layer. The cube expansion technique is interesting because each cube obtained indeed removes at last one cube in the cover. In (SHIN; GUPTA, 2010), it is applied an expansion approach being said that there is no guarantee to the best approximation attains by using it. Nevertheless, we could not find an example that confirms it, and a formal proof is out of the scope of this work.

The enumeration phase execution results in a set of cubes that inserts at most the error threshold. In this way, there are multiple cubes which add fewer errors than the threshold. As inserting a cube with less error than the threshold can be a waste of optimization, it is interesting to insert a set of cubes that respect the error threshold instead of only one cube. Hence, finding an approximate cover that minimizes the number of

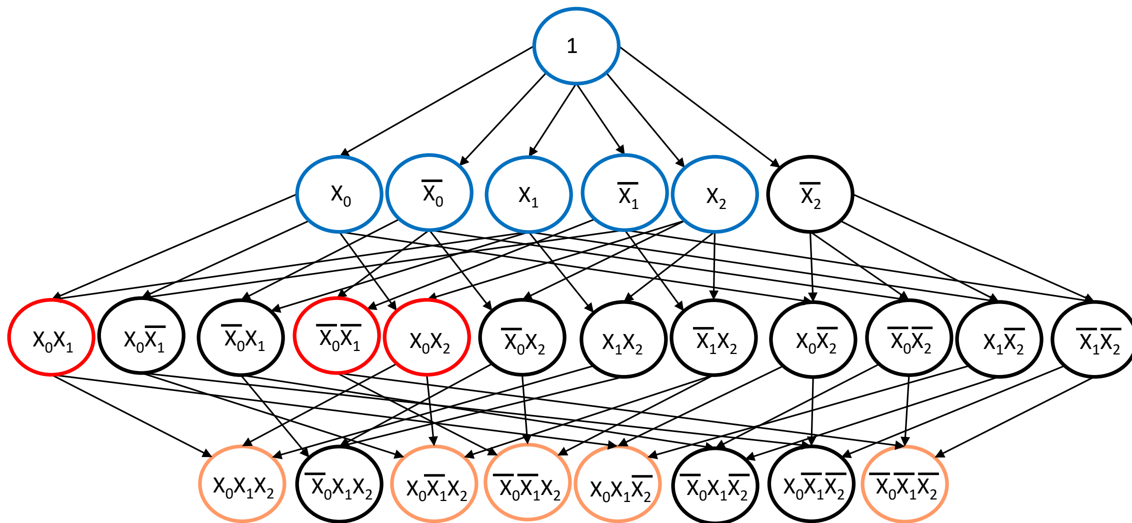


Figure 3.7 – Hasse diagram of 3-inputs 1-output function with the cubes that an expansion technique can enumerate over the cubes in $\mathcal{C}$ highlighted in blue.

literal depends on selecting the set of cubes that, when inserted, leads to the greatest reduction on cover literal count. The sets of cubes are obtained by combining the cubes until no new set can be found. The main problem of this combination is that the number of cubes grows exponentially with the number of allowed errors.

After obtaining the sets of cubes that can be inserted, it is necessary to calculate the literal reduction for each set. In (SU et al., 2020), it is presented a procedure that efficiently estimates the literal reduction. It consists of identifying which cubes in the cover can be removed when a given set of cubes are inserted. A cube can be removed from the cover when there is no minterm only covered by this cube. This identification consists of finding the cover cubes where all minterms exclusively covered by it are also covered by the cubes to be inserted. In (SHIN; GUPTA, 2010), it is applied an expansion approach that stores all cover cubes that result in the same expanded cubes. As the expansion of a cube generates an expanded cube that covers the original one, when the expanded cube is inserted in the cover, the original cube can be removed. The number of reduced literals is equal to the difference between the literal count in the removed cubes and the inserted cubes.

In (ZOU; QIAN; HAN, 2015), the proposed approach does not present a specific combination and literal reduction estimation procedures as it chooses the cubes to be inserted in a dynamic programming approach. The selection of the cubes to be inserted is based on a metric that considers the cube size and the number of literal. Nevertheless, this metric has a weak correlation to the literal reduction when the set of cubes is added into the original cover, and not always is proportional to the literal reduction.

As stated at the end of Section 3.5, all 2L-ALS methods use truth tables to calculated the inserted error. It is valid for these three works, but neither applies an explicit error calculation method to verify the ER constraint. These methods control the number of inserted errors by verifying how many input combinations are modified in the truth table when inserting a cube.

In this work, we propose a 2L-ALS approach that approximates a cover taking into account the ER metric as the constraint. Our approach seems to be the first one in applying both cube insertion and removal techniques to approximate a cover. The basic flow for cube insertion technique presented in this subsection is adopted in our work, with the addition of a cube removal approach. The details of our approach are shown in the next section.

4 PROPOSED 2L-ALS METHOD

This work focuses on the approximation of a cover considering ER as error constrain. As shown in Subsection 3.6.2, the works which address this problem apply the cube insertion technique to approximate the cover. These works only consider this technique based on the experiment presented in (SHIN; GUPTA, 2010), which shows that the cube insertion technique leads to a better approximation than the cube removal technique. Note that any of these works consider using both techniques together.

Thus, a motivational example about using both techniques together is shown in Figure 4.1, which considers the approximation of the cover presented in 4.1a with the possibility to insert two errors at most. In Figure 4.1b and in Figure 4.1c is shown the approximation using the cube insertion technique. In Figure 4.1d and in Figure 4.1e is shown the approximation using the cube removal technique. In Figure 4.1f and in Figure 4.1f is shown the approximation using both techniques together. The resulting covers show that using both techniques together leads to a more significant optimization in the number of literals when compared to their application alone. Hence, the use of both techniques together in an ALS method seems to be promising.

This work presents a 2L-ALS method that applies the cube insertion and cube removal techniques to approximate a cover with ER as error constraint. At first, in this chapter, is is presented some details of the Su's cube insertion procedure that are used on the proposed approach. Next, it is presented the primary data structure used to develop the proposed method. Then, a general description of the proposed method is shown, mentioning the main stages of the algorithm and the data structure of applications. Afterwards, the main stages are detailed, focusing on the cube insertion and the cube removal procedures. Besides the proposed method, the use of Espresso as a tool for post-processing is also discussed. At the end, a time complexity analysis for the proposed method is shown.

4.1 Su's Cube Insertion Approach

In (SU et al., 2020), Su *et al.* present a heuristic search method to solve the 2L-ALS problem taking into account ER constraint. This work can be considered as the state-of-the-art method in the subject.

The main goal of the Su's approach is to identify the set of input combinations for 0-to-1 output complement (SICC) that maximize the literal count reduction on an

x_2 \ x_0x_1	00	01	11	10
0	0	1	1	0
1	0	0	1	1

(a) Karnaugh map representation of the original cover. The SOP expression of this cover is $x_0 * x_2 + x_1 * !x_2$

x_2 \ x_0x_1	00	01	11	10
0	0	1	1	1
1	0	1	1	1

(b) Minterms to be complemented using cube insertion approach

x_2 \ x_0x_1	00	01	11	10
0	0	1	1	1
1	0	1	1	1

(c) Approximate cover by cube insertion approach. The SOP expression of this cover is $x_0 + x_1$

x_2 \ x_0x_1	00	01	11	10
0	0	0	0	0
1	0	0	1	1

(d) Minterms to be complemented using cube removal approach

x_2 \ x_0x_1	00	01	11	10
0	0	0	0	0
1	0	0	1	1

(e) Approximate cover by cube removal approach. The SOP expression of this cover is $x_0 * x_2$

x_2 \ x_0x_1	00	01	11	10
0	0	0	1	1
1	0	0	1	1

(f) Minterms to be complemented using both approaches

x_2 \ x_0x_1	00	01	11	10
0	0	0	1	1
1	0	0	1	1

(g) Approximate cover by both approaches. The SOP expression of this cover is x_0

Figure 4.1 – Approximate covers examples comparing cube insertion, cube removal and both approaches together.

approximate cover. It is similar to select the set of EICs that results in the most compact cover. They propose an SICC-cube tree (SCT) data structure, which groups a set of EICs to a set of cubes that depends on these EICs to be inserted into the cover. It comprises a two-level tree where the root contains the EICs and the leaves represent the cubes to be added into the cover.

Two conditions must be satisfied to ensure that SCT leaves lead to the optimization of the literal count. Firstly, at least one cube must be removed from the cover when a new cube is inserted. Secondly, the literal count in the removed cubes must be greater than the literals present in inserted cube.

Their initial task enumerates all possible multiple-output cubes of a function through the Hasse diagram structure. These cubes are used to build a set of SCTs. In the next, the SCTs are combined because there are some with fewer errors than the maximum number allowed. After that, it is necessary to select the SCT that reduces the greatest number of literals.

A straightforward way to calculates the literal reduction in a given SCT is by using the Espresso tool (BRAYTON et al., 1984), taking into account the EICs on the root as *don't cares* to obtain an approximate cover. The calculation of the literal reduction with Espresso presents a precise result, but the impact on the runtime is quite significant. Hence, a procedure that avoids the use of Espresso for estimating such a reduction is presented.

The procedure to predict the literal reduction on an SCT comprises main three steps. As the insertion of leaf cubes into the cover does not guarantee a reduction in literal count, it first identifies the set of cubes that may be removed when the leaf cubes are inserted. Moreover, inserting all leaf cubes may increase the cover literal count. Thus, it identifies the set of leaf cubes necessary to be inserted before removing the first set of cubes. Finally, it calculates the literal reduction between the sets of cubes removed and inserted.

In (SU et al., 2020), the authors present four speed-up techniques to extend the application of their approach to large circuits.

1. As the basic algorithm time complexity grows exponentially with the NoE, the errors allowed for each execution is limited to two, so generating partial approximate covers. All partial covers are approximated again until the accumulated NoE reaches the threshold allowed.
2. With the first speed-up technique, an exponential quantity of partial approximate

covers is created, so impacting the final runtime. In order to reduce the number of partial covers, only the two expressions with the fewest number of literals are approximate again for a given partial NoE.

3. To reduce the number of combined SCTs, only a subset of all generated SCTs are taken into account. At first, it estimates the literal count of all SCTs without combining them. In the next, for combining two SCTs, the first SCT must be on 25% with the fewest number of literals whereas the second SCT must be on 80% of these ones.
4. The treatment of all cubes present on the Hasse diagram implies in a high computational cost. In order to reduce such a cost, they only take the cubes on the diagram that are the parents of the cubes on the cover, since it is improbable that other than a parent of a cover cube inserts less than two errors.

4.2 Data Structure

This work uses three main data structures: the first one stores the cover during the approximation, the second stores and organize the cubes which can be inserted on the cover, and the third contains information used to modify the original cover, preventing the creation of a new cover to each approximation.

4.2.1 Cover

As discussed previously, a cover comprises the cube prime implicants needed to cover all minterms of a given function. During this chapter, the cube prime implicants are called only by cubes. A map is used to stores each cube from the cover and their respective *covered once minterms* (COM). As COMs are used in many operations, storing them with the respective cube accelerates these operations.

Besides the cover cubes, the adopted data structure also uses a map to store the minterms covered and the cubes covering each minterm. Although it seems to be redundant, storing cubes and minterms in that way prevents operations related to find the cubes that cover a minterm, which could be very time-consuming.

This structure is modified when the cover has cubes inserted or removed. In that way, updating such a structure computational cost is smaller than computing the COMs

and cubes that cover a minterm when necessary.

4.2.2 SICC-cube tree (SCT)

To store and organize the cubes which can be inserted on the cover during the approximation the SICC-cube tree (SCT) data structure proposed in (SU et al., 2020) is used.

As said before, an SCT comprises a two-level tree where the root contains a set of erroneous input combinations (EIC), and the leaves contains new cubes that may be inserted in the cover when the input combination in the root are approximated.

When the leaves cubes are inserted in the cover, the number of errors inserted is equal to the number of EICs in the root. The leaf cubes are sorted because they will be used in a greedy optimization. This sort considers their size and the number of cubes removed from the cover when the cube is inserted, in decreasing order.

4.2.3 Solutions

The approximation procedure generates multiple partial solutions, creating a new cover for each partial solution. As the cover data structure contains a large quantity of information, storing multiple cover during the approximation is unfeasible. Thus, instead of storing a cover, only the difference between the approximated cover and the original one is stored.

The data structure solutions stores these differences. The main differences between the covers are the cubes that they contain. In that way, this data structure contains two sets of cubes, the cubes to be inserted and the cubes to be removed. A set of EICs introduced by the approximation is also stored. In the last step, the number of literal reduced by the approximation is stored.

4.3 General Description

The problem addressed in this method is to identify an approximate cover C' with the fewest literal count for a given original cover C and an ER threshold TE. Instead of using a percentage TE during the approximation, the method limits the number of

errors (NoE) e inserted, which is given by $e = TE * 2^n$. The presented method is a two-phased method that utilizes a cube insertion approach and then a cube removal approach to synthesize an optimized cover $\mathcal{C}'$. The cube insertion approach is made first because of its potential to generate better solutions. A problem here is how many NoE each approach is allowed to insert in order to generate a minimized cover and not exceed the NoE threshold. The simple answer to this problem is that it depends on the original cover, and it is hard to define a value for any expression. In that way, the proposed method considers all values of e_i and e_r in the range (0,e) that satisfy the equality $e = e_r + e_i$, where e_i and e_r are the error thresholds allowed to the cube insertion approach and to cube removal approach, respectively. The strategy is to apply a cube insertion approach that generates partial solutions with crescent NoE and then apply the cube removal approach over these partial solutions with the remaining NoE. In that way, each cube removal execution is allowed to insert $e - e_i$ NoE. Works that utilize cube insertion procedures usually generate partial solutions while deriving the approximate solution $\mathcal{C}'$.

The generation of partial solutions is done by limiting the NoE that each cube insertion execution can insert. Each execution is allowed to insert two errors, and results in two partial solutions with one and two errors. In that way, when the cover $\mathcal{C}$ is approximated, two partial solutions are generated with one and two errors. Then the partial solution with one error is approximated, and two partial solutions with 2 and 3 errors concerning $\mathcal{C}$ are created. This process continues to every generated partial solution until it reaches the NoE threshold. This behavior is based on the first speed-up technique presented on Section 4.1. An example of this process for the NoE threshold of 4 is illustrated in Figure 4.2, where each square represents a solution and its value the solution NoE.

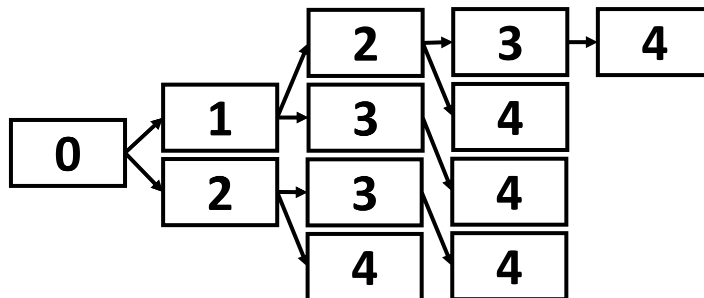


Figure 4.2 – The behavior of the iterative partial solutions generation, considering a limit of 4 NoE. Each partial solution generates two new partial solutions, increasing NoE in 1 and 2 until it reaches the NoE limit.

Using this limitation on the NoE inserted by each cube insertion execution generates a high partial solution quantity. In fact, the number of partial solutions for a given NoE follows the the Fibonacci sequence. This behavior for an NoE equals to 8 is shown

in Figure 4.3. As the cube insertion procedure is executed over each generated partial solution, its high quantity impacts the method scalability. Instead of considering all generated partial solutions, the method selects only the two best partial solutions for each NoE. With this greedy selection, it will be generated at most four partial solutions with each NoE. This greedy selection is based on the second speed-up technique presented on Section 4.1.

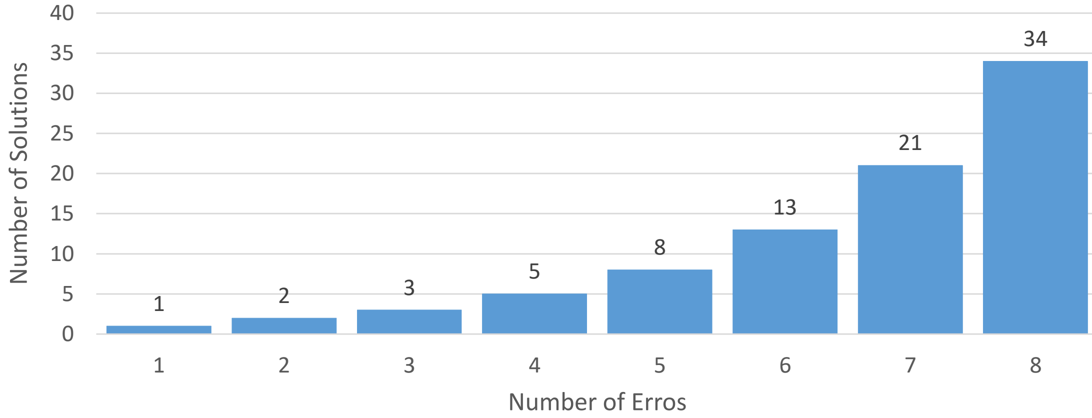


Figure 4.3 – Graph showing the number of solution generated with each NoE for an approximation with a limited to 8 NoE, totalizing 87 partial solutions.

Algorithm 1: Proposed Two-Level ALS Method

Input: A simplified cover $\mathcal{C}$ and ER threshold TE
Output: An approximated cover $\mathcal{C}$'

1. e $\leftarrow TE * 2^n$;
2. $\text{Solutions}_{0\rightarrow e}$ sets of solutions;
3. $\text{Solutions}_0 \leftarrow \oslash$ (empty solution);
4. **for** $i \leftarrow 0$ *to e* **do**
5.   topS $\leftarrow$ two best results in $\text{Solutions}_\text{i}$;
6.   **for** *each Solution s in topS* **do**
7.   ModifyCover($\mathcal{C}$,s);
8.   (s1,s2) $\leftarrow$ CubeInsertion($\mathcal{C}$,2,s);
9. $\text{Solutions}_\text{i+1} \leftarrow \text{Solutions}_\text{i+1} \cup$ s1;
10. $\text{Solutions}_\text{i+2} \leftarrow \text{Solutions}_\text{i+2} \cup$ s2;
11.   s3 $\leftarrow$ CubeRemoval($\mathcal{C}$,e-i,s);
12.   sMax $\leftarrow$ max(s1,s2,s3);
13.   **if** *sMax* $>$ *best* **then** best$\leftarrow$sMax;
14.   RestoreCover($\mathcal{C}$,s);
15.   **end**
16. **end**
17. return espresso(ModifyCover($\mathcal{C}$,best));

The overall flow of our proposed method is illustrated in Algorithm 1. In line 2, the set of $Solutions$ comprises all the partial solutions obtained, whereas the set of

$Solutions_i$ comprises the partial solutions with i errors inserted. To modify the cover $\mathcal{C}$ taking into account a solution s, it is applied the *ModifyCover* function, corresponding to the line 7 in Algorithm 1, which inserts and removes cubes in $\mathcal{C}$. On the other hand, the *RestoreCover* function, in the line 14, undo these modifications, inserting the cubes that have been removed and removing the ones that have been inserted. The insertion procedure, in the line 8, returns two solutions (*s1* and *s2*) that are stored in $Solutions_i + 1$ and $Solutions_i + 2$, in the lines 9 and 10, respectively. In the removal procedure, in the line 11, for a given NoE e, it is allowed to insert $e - i$ errors, returning the solution *s3*. The best solution is then updated with the one that contains the most significant literal reduction. At the end, the best solution is applied to $\mathcal{C}$, which is optimized by the Espresso tool (BRAYTON et al., 1984).

4.4 Cube Insertion Procedure

In this section is described the cube insertion procedure, detailing how it works. Our cube insertion procedure are based on the one presented by Su's (SU et al., 2020), presented in Section 4.1. The main idea is to generate SCTs from cubes that do not exceed the threshold number of EICs and then select the SCT with the most significant literals reduction. Algorithm 2 presents the cube insertion flow.

Algorithm 2: CubeInsertion Procedure

Input: A simplified cover $\mathcal{C}$ and the current solution s
Output: Two solutions with error 1 and 2

```
1 trees ← generateSCT(C,s.EIC);
2 augment(trees);
3 (s1,s2) ← combineAndEstimate(C,trees);
4 s1 ← updateSolution(s1,s);
5 s2 ← updateSolution(s2,s);
6 return (s1,s2);
```

In the line 1, it generates all SCTs. This procedure is detailed in Subsection 4.4.1.

For SCTs *sct1* and *sct2* containing 1 and 2 EICs in the root, respectively, if the *sct2* root comprises the *sct1* root, the leaves of the *sct1* are inserted in the *sct2* leaves. This updated *sct2* is called an augmented SCT, and this process is done in the line 2.

The line 3 performs the combination of SCTs and the estimation of literal reduction. This procedure is detailed in Subsection 4.4.2. The two solutions that reduce more literals with 1 and 2 NoE are returned and stored in *s1* and *s2*.

The solutions *s1* and *s2* are updated in the lines 4 and 5. This update consists of adding the cubes inserted and removed within solution *s* into solutions *s1* and *s2*, estimating the new literal reduction, and updating the EICs. At the end, this procedure returns the solutions *s1* and *s2*.

In multilevel ALS methods, it is usually necessary to estimate the error for each possible approximation by using SAT, BDD or simulation, for instance. The proposed approach does not need continuous error checking for each possible approximation because the error control is done by the number of EICs in the SCT root. It is essential to mention that it is only possible because all input-output relationship is known during the approximation.

4.4.1 SICC Cube-Tree Generation

Introduced in Subsection 4.2.2, the SICC-cube tree (SCT) is a data structure applied to enumerate the cubes that can be inserted in the cover for a set of EICs. To enumerate the cubes to be inserted in the cover, we use a cube expansion approach allowing only one literal removing. In other words, it is only considered the cubes on the Hasse diagram that are direct parents of the cover cubes. As the NoE for each execution is limited to two, the set of EICs is at most two input combinations. It is improbable that cubes found many levels above the cover cubes in the Hasse diagram inserts two or few errors, so using the proposed enumeration strategy seems appropriate.

For an enumerated cube to be inserted as an SCT leaf, it must satisfy some conditions. At first, the cube insertion must need the insertion of more than zero and at most two new EICs. If a cube needs zero new EICs, it does not approximate the cover and is ignored. It is also necessary that the number of literals on the expanded cube are equal to or greater than the number of literals removed by its insertion. This number of literals removed is estimated considering the cubes that all COMs are covered by the expanded cube. This verification prevents the insertion of cubes that do not optimize the literals number.

When a cube satisfies these restrictions, it can be inserted as a leaf in an SCT, which has the EICs needed by this cube as root. If this SCT does not exists, it is constructed. The general flow of this procedure is shown in Algorithm 3.

Algorithm 3: generateSCT Procedure

```
   Input: a cover 𝒞 and a set of used EICs eics
   Output: A set of SCT
 1 set of SCT scts;
 2 for each cube C in 𝒞 do
 3 |   for each expanded cube C_e from C do
 4 |   |   EIC ← getEIC(C_e, 𝒞, eics);
 5 |   |   if EIC.size in interval [0,2) then
 6 |   |   |   if C_e removes more literals than it inserts then
 7 |   |   |   |   if exists SCT s with root EIC in scts then
 8 |   |   |   |   |   insert C_e as leaf of s;
 9 |   |   |   |   end
10 |   |   |   |   else
11 |   |   |   |   |   create new sct s with EIC as root;
12 |   |   |   |   |   insert C_e as leaf of s;
13 |   |   |   |   |   insert s in scts;
14 |   |   |   |   end
15 |   |   |   end
16 |   |   end
17 |   end
18 end
19 return scts;
```

4.4.2 Combine and Estimate

After getting all feasible SCTs, it is needed to verify which two SCTs with one and two errors lead to the most significant minimization of literals count. Initially, it estimates how many literals are reduced by each SCT. Then, the SCTs with one EIC are combined two by two, generating more SCTs with two EIC. The combination of the two SCTs is done by merging the root EICs and the cubes on the leaves. The literal count estimation is done over the combined SCTs. The generation of all possible combinations may lead to a significant impact on performance. In that way, only subsets of SCTs are selected to be combined, based on its literal reduction. For the two SCT sct1 and sct2, sct1 must be in the 25% best SCTs and sct2 in the 80% SCTs. By making this selection, the number of combined SCTs is reduced by 80% when compared to the combination of all SCTs. Algorithm 4 presents the flow explained above.

To estimate the literal reduction it is necessary to identify which cubes are inserted in the cover and removed from the cover. To identify the cubes to be removed, we obtain a set with all minterms covered by the leaves cubes in the SCT. Then, if a cover cube has all its COMs in the minterms set, it is removed from the cover. The idea is that if a cube

Algorithm 4: combineAndEstimate

```
   Input: a cover 𝒞 and a set of SCT scts
   Output: Pair of solutions with 1 and 2 NoE
1  for each SCT sct in scts do
2  |  sct.reduction = estimateLiterals(sct, 𝒞);
3  end
4  scts_1 ← SCTs in scts with 1 root's EIC;
5  scts_2 ← SCTs in scts with 2 root's EIC;
6  sort scts_1 in decrescent order of literal reduction;
7  for each SCT sct_a in the first 25% positions of scts_1 do
8  |  for each SCT sct_b in the first 80% positions of scts_1 do
9  |  |  sct_c ← combine(sct_a, sct_b);
10 |  |  sct_c.reduction = estimateLiterals(sct_c, 𝒞);
11 |  |  insert sct_c in scts_2;
12 |  end
13 end
14 best_1 ← SCT with greater literal reduction in scts_1;
15 best_2 ← SCT with greater literal reduction in scts_2;
16 return (best_1,best_2);
```

has all COMs covered, this cube is redundant and can be removed without modifying the Boolean function. Thus, when all leaves cubes are inserted, these cubes become redundant and can be removed. After each cube removal, all COMs are updated. This update is done because when we remove a cube, other COMs may change. If it is not done, it can lead to an erroneous cube removal and inserting more errors than expected. In Subsection 4.4.1, a literal estimation that does not update the COMs is used when a cube may be inserted in an SCT because it is an initial estimation. And modifying the cover for each possible cube could be computationally expensive. The removed cube COMs are stored to be used later in the process.

After getting the cubes to be removed, the next step is to select the cubes to be inserted. The SCT leaves are used to select the removed cubes, and they need to be inserted in the cover to guarantee that all uncovered COM are covered again. Nevertheless, if all leaf cubes are inserted, it may result in new redundant cubes. Hence, in this step, the main objective is to obtain the subset of SCT leaves cubes necessary to cover the uncovered COMs without generating new redundant cubes.

In Subsection 4.2.2, it was defined that the cube leaves are sorted in decreasing order based on its size and number of removed cubes when this cube is inserted. In that way, the set of cube leaves are traveled in order, with the cubes that cover more minterms and removes more cubes at the beginning. If a cube covers at least one of the uncovered

COMs, it is inserted in the cover, and the covered ones are marked. This procedure is done until all COMs are covered.

In special cases, even after this procedure, some redundant cubes are inserted in the cover. These cases are solved by verifying in the inversed order of insertion if a cube has no COMs. As mentioned before, if a cube has no COMs, it can be removed from the cover.

The number of literals that a given SCT removes is equal to the number of literals in removed cubes minus the number of literals in the obtained subset of leaves cubes. At the end, the number of literals is returned.

Algorithm 5 presents the procedure to estimate the literal reduction for an SCT. The minterms covered by the SCT leaves are got in the lines 4-6. The cubes to be removed are obtained in the lines 7-13. The subset of cubes in SCTs inserted in the cover is obtained in the lines 14-20. In the lines 21-25 are verified whether there are redundant cubes. In the line 26 are calculated the literal count reduction, which is returned at the end.

4.5 Cube Removal Procedure

The cube removal procedure is a greedy algorithm that selects the cube with the highest score. The score is the ratio between the number of literals and the number of EICs of the cube to be removed. While removing a cube, its EICs is given by the number of minterms covered only by this cube which is not contained in the newEIC set. Algorithm 6 shows the flow of this procedure.

In the loop in line 3, the cube is chosen to be removed. For that, the EICs are obtained in the line 4 and the score in the line 5. If the score is greater than the current best, it is stored in the line 7. In the line 12, the best cube is removed from $\mathcal{C}$, updating the COMs of other cubes. Then, the set of the removed cube, the set of EICs, and the errors number are updated.

When the allowed NoE is reached, no more cubes can be removed, and the main loop ends. At the end, the cubes are re-inserted in $\mathcal{C}$, and the solution s3, comprising the removed cubes, the inserted EICs, and an updated literal reduction count is returned.

Algorithm 5: estimateLiterals

```
   Input: An SCT sct and a cover C
   Output: The SCT literal reduction
 1 set of minterms leavesMinterms;
 2 set of COMs removedCOMs;
 3 set of cubes removedCubes and insertedCubes;
 4 for each cube c in sct leaves do
 5 |  insert minterms covered by c into leavesMinterms;
 6 end
 7 for each cube c in C do
 8 |  if leavesMinterms contains all COMs of c then
 9 |  |  insert c COMs in removedCOMs;
10 |  |  remove c from the cover and update the COMs;
11 |  |  insert c in removedCubes;
12 |  end
13 end
14 for each cube c in sct leaves do
15 |  if c covers at least one COM in removedCOMs then
16 |  |  insert c in insertedCubes;
17 |  |  remove from removedCOMs the covered COMs;
18 |  |  insert c in C;
19 |  end
20 end
21 for each cube c in insertedCubes in reverse insertion order do
22 |  if c do not have any COM then
23 |  |  remove c from C;
24 |  end
25 end
26 literalReduction ← number of literals in removedCubes − number of literals
    in insertedCubes;
27 return literalReduction;
```

Algorithm 6: cubeRemoval Procedure

```
   Input: A simplified cover 𝒞, an NoE threshold e and the current solution s
   Output: A solution s3 with at most e errors
 1 error ← e, newEIC ← s.EIC;
 2 while error > 0 do
 3   for each cube C in 𝒞 do
 4     cubeEIC ← getCubeEIC(C, 𝒞, newEIC);
 5     score ← LitCount(C)/#cubeEIC;
 6     if score > bestScore and error ≥ #cubeEIC then
 7       bestScore ← score;
 8       bestCube ← cube;
 9       bestEIC ← cubeEIC;
10     end
11   end
12   removeCubeFromCover(bestCube, 𝒞);
13   removedCubes ← removedCubes ∪ bestCube;
14   newEIC ← newEIC ∪ bestEIC;
15   error ← error - #bestEIC;
16 end
17 insertCubes(𝒞, removedCubes);
18 s3 ← updateSolution(removedCubes, newEic, s);
19 return s3;
```

4.5.1 Speed-Up Optimization

The cube removal procedure is relatively straightforward and simpler than the cube insertion procedure. Besides that, the cube removal procedure does not have a limit in the NoE inserted per execution, impacting the proposed method performance. A way to accelerate this procedure is to prevent the score recalculation in cubes that the EICs were not modified, i.e., when the score does not change the value. Moreover, the current scores are sorted in decrescent order, with the first cube being removed in each iteration.

The removeCubeFromCover function is modified to return the cubes in which the EICs are modified. Only these cubes have their score recalculated. To link the cubes and the scores it is used two Maps. The Map cubesToScore stores the relation between the cubes in the cover and their scores. The Map scores is an ordered map that maintains the values sorted in decrescent order and stores the relation between the score and the cubes to be removed. The Map cubesToScore may seem redundant, but its use simplify the score update.

The optimized cube removal procedure is shown in Algorithm 7. The loop in Lines 4 -9 calculates the score for each cube in the SOP and stores it in Maps cubesToScore

and scores. Then, inside the loop in Lines 10-22 are performed the cube removal until the error limit is reached. It starts selecting the first entry of the Map scores as bestScore. This entry in scores and the respective entry in the cubesToScore are removed. If the number of EICs in the selected cube is less than the available error, the cube can be removed. When a cube is removed, the set of cubes that had their EICs modified is returned and then used to update the scores. The rest of the Algorithm has the same behavior as the Algorithm 6.

Algorithm 7: CubeRemoval Procedure Optimized

```
   Input: A simplified cover 𝒞, an NoE threshold e and the current solution s
   Output: A solution with at most e errors
 1 error ← e, newEIC ← s.EIC;
 2 Ordered Map scores;
 3 Map cubeToScore;
 4 for each cube C in 𝒞 do
 5 |  cubeEIC ← getCubeEIC(C,𝒞,newEIC);
 6 |  score ← LitCount(C)/cubeEIC.size;
 7 |  insert in scores the entry (score,C);
 8 |  insert in cubeToScore the entry (C, score);
 9 end
10 while error > 0 do
11 |  bestScore ← first score in scores;
12 |  remove first score from scores;
13 |  remove bestScore.cube from cubesToScore;
14 |  cubeEIC ← getCubeEIC(bestScore.cube,𝒞,newEIC);
15 |  if cubeEIC.size < error then
16 |  |  modifiedCubes ← removeCubeFromCover(bestCube,𝒞,newEIC);
17 |  |  updateScores(cubeToScore, scores, modifiedCubes, newEIC, 𝒞);
18 |  |  removedCubes ← removedCubes ∪ bestCube;
19 |  |  newEIC ← newEIC ∪ bestEIC;
20 |  |  error ← error - sizeof(bestEIC);
21 |  end
22 end
23 insertCubes(𝒞,removedCubes);
24 s3 ← updateSolution(removedCubes,newEic,s);
25 return s3;
```

The updateScore procedure is shown in Algorithm 8. For each cube with its EICs modified, the set of modified EICs is obtained, and a new score is calculated considering the number of literals and these new EICs. The entry of the maps with the old score for the cube is removed, and a new one with the updated score is inserted. The entry in the cubesToScore has the respective score updated. This Algorithm shows how the cubesToScore facilitates the scores update.

Algorithm 8: updateScore

```
   Input: A Map cubeToScore, an Ordered Map scores, a set of cubes
          modifiedCubes, a set of EICs eics, a cover 𝒞
1  for each cube C in modifiedCubes do
2  |   cubeEIC ← getCubeEIC(C,𝒞,eics);
3  |   newScore ← LitCount(C)/cubeEIC.size;
4  |   oldScore ← cubeToScore at cubeEIC;
5  |   remove scores at oldScore;
6  |   insert in scores the entry (newScore,C);
7  |   update cubeToScore at C with newScore;
8  end
```

4.6 Post-Processing Tools

At the end of Algorithm 1, which presents the general flow of the proposed method, the best solution is optimized using the Expresso tool. This optimization is needed because the insertion and removal of cubes modify the Boolean function and may exist a new cover with fewer literals than the one obtained.

Besides optimizing the generated cover with Espresso, it is also possible to modify the cover and optimize it as an ISBF. An ISBF can be seen as a cover where some input combination does not have a defined Boolean value, and both 0 and 1 are valid, calling this input a *don't care*. When the Espresso is used to optimize an ISBF, it selects the output value for the don't care inputs that lead to a cover with fewer literals.

To modify a cover into an ISBF, we use the concepts of EIC and ER. An EIC is an input combination that results in one or more erroneous output bits, and the ER metric is the number of EICs divided by the number of input combinations. Note that even if multiple output bits have an erroneous value for an EIC, the ER is not changed. Thus, we can assume that all outputs bits resulted from an EIC are don't care because they do not affect the ER value.

In that way, the input of the Espresso is the cover obtained by the approximation and each EIC generated is set as don't cares, allowing the Espresso to manipulate their outputs in order to optimize the final cover. It is interesting to observe that depending on the choices made in the Espresso optimization, the output for some EICs may be set as the original output value, so reducing the final cover ER.

4.7 Time Complexity Analysis

This section presents the time complexity analysis of the proposed method. The Algorithm 1 presents the general flow of the method, so the analysis starts in this algorithm and then focuses on each of its functions. Algorithm 1 has two main loops, the first with e+1 iterations and the second with only two. The inner loop contains three main tasks: the cover modification, the cubeInsertion procedure, and the cubeRemoval procedure. As the cover modification has significantly less time impact than the approximation procedures, it has been ignored in this analysis. For the sake of simplicity, we are omitting the Espresso complexity.

The combineAndEstimate task in the cubeInsertion procedure is the most time consuming one. The most expensive phase of this procedure is to combine two by two the SCTs with one EIC on the root and estimate their literal count reduction. To generate the SCTs, the cubes on the SOP are expanded. As the expansion generates a new cube for every literal in a cube, the number of expanded cubes is equal to the number of literals in the SOP, represented by *L*. The worst-case number of SCTs with one EIC is reached when each expanded cube generates one of them. Thus, the number of SCTs that have their literals estimated is up to $O(L^2)$. The literal estimation depends on obtaining the covered minterms of each leaf cubes. As the number of covered minterms by a cube is at most $O(m * 2^n)$, where n and m are the number of inputs and outputs of the function, respectively, the worst-case time complexity of the cubeInsertion procedure is $O(L^2 * m * 2^n)$.

The cubeRemoval procedure, in turn, estimates the score of removing each cube, represented by *C*, and removes the one with more score until the limit error is reached. The score depends on the number of EICs and cube literals. As obtaining the EICs relies on hash structures, its time complexity can be taken as constant. To obtain literal count, the cube are iterated $O(n + m)$ times. Therefore, the worst-case time complexity of the cubeRemoval procedure is $O(e * C*(n + m))$.

The complete worst-case time complexity of the proposed method is $O(e*(L^2 * m * 2^n + e * C\ (n + m)))$.

5 EXPERIMENTAL RESULTS

The proposed algorithms have been implemented in C++ programming language. Our experiments have been carried out over the IWLS'93 benchmark suite (MCELVAIN, 1993), in a laptop with a dual-core i7-7500U processor @ 2.70GHz and 16GB of RAM. The Espresso tool has been applied to optimize the approximate cover to generate results shown in Table 5.1, and both approximate cover and related ISBF version are optimized as shown in Table 5.2, which presents the best solution between them. In Section 4.6, the use of each optimization is discussed.

5.1 Comparison to the State-of-Art Approach

The Su's method, proposed in (SU et al., 2020), can be considered as the state-of-the-art for 2L-ALS, so it has been taken into account herein as the golden reference. Unfortunately, the code of Su's approach is not publicly available, being the values presented in this section obtained from the Su's work. Also, we do not have access to a computer with the exact specifications used to generate their experimental results. To present a fair comparison, the CPU used in this experiment has a similar performance to their CPU, even though it was launched more recently. The experiments consider the same circuits and NoE threshold as in Su's work in order to allow a fair comparison. Thus, the designs have more than 6 and fewer than 20 inputs, and the NoE threshold equals 16.

Table 5.1 shows the comparison results between Su's method and our proposed approach. Column 1 presents the name of the circuits, as well as the number of inputs (i) and the outputs (o). Column 2 shows the number of literals of the original circuits, whereas column 3 and column 4 present the number of literals of the approximate circuits presented in (SU et al., 2020) and obtained from our method, respectively. Column 5 shows the literal reduction rate between the literal count from the approximate cover generated by our approach and the one presented in (SU et al., 2020). Column 6 shows the number of cuber of the original circuits, whereas column 7 and column 8 present the number of cubes of the approximate circuits presented in (SU et al., 2020) and obtained from our method, respectively. Column 9 shows the cube reduction rate between the cubes count from the approximate cover generated by our approach and the one presented in (SU et al., 2020). Column 10 and column 11 present the runtime for both methods.

Our method presented better results for all circuits treated, except the *b12* one

Table 5.1 – Comparison to Su's approach (SU et al., 2020) in IWLS'93 benchmark suite considering NoE threshold of 16.

Circuit	Literals				Cubes				Time(s)	
	Orig.	Su's	Ours	Ours/Su's	Orig.	Su's	Ours	Ours/Su's	Su's	Ours
con1 i:7;o:2	32	32	24	**0.75**	9	9	7	**0.77**	0.38	0.02
rd73 i:7;o:3	903	578	556	**0.96**	127	88	88	1.00	1.48	1.37
inc i:7;o:9	198	156	125	**0.80**	30	25	20	**0.80**	0.49	0.18
5xp1 i:7;o:10	347	235	202	**0.85**	65	49	41	**0.83**	0.72	0.48
sqrt8 i:8;o:4	188	98	83	**0.84**	38	22	21	**0.95**	0.58	0.24
rd84 i:8;o:4	2070	1578	1511	**0.95**	255	218	209	**0.95**	6.52	4.51
misex1 i:8;o:7	96	96	77	**0.80**	12	12	10	**0.83**	0.50	0.02
clip i:9;o:5	793	588	584	**0.99**	120	93	93	1.00	1.99	1.36
apex4 i:9;o:19	5419	5040	5024	**0.99**	436	421	418	**0.99**	109	31.9
sao2 i:10;o:4	496	231	165	**0.71**	58	29	22	**0.75**	2.48	1.47
ex1010 i:10;o:10	2718	2693	2636	**0.97**	284	283	278	**0.98**	14.30	2.05
alu4 i:14;o:8	5087	4904	4847	**0.98**	575	562	559	**0.99**	298	14.06
misex3 i:14;o:14	7784	7446	7242	**0.97**	690	656	635	**0.96**	693	11.99
table3 i:14;o:14	2644	2459	2347	**0.95**	175	165	159	**0.96**	513	5.47
misex3c i:14;o:14	1561	1239	1115	**0.89**	197	163	153	**0.93**	252	19.29
b12 i:15;o:9	207	207	207	1.00	43	43	43	1.00	249	1.69
t481 i:16;o:1	5233	5105	4975	**0.97**	481	473	463	**0.97**	1570	3.41
table5 i:17;o:15	2501	2410	2270	**0.94**	158	154	147	**0.95**	7868	22.07
Average	2126	1949	1888	**0.90**	208.5	192.5	186,9	**0.92**	643	6.75

that could not optimize by both approaches. The circuits *con1*, *misex1* and *b12* could not be approximated by the Su's method, as presented in (SU et al., 2020), as it does not have SCT with size equals to 1 or 2. On the other hand, *con1* and *misex1* have been approximated by our method due to the cube removal phase. Results for literal and cubes reduction are similar. Our method presented results better or equal to Su's approach in every circuit for both cubes and literals.

Moreover, the proposed method presents a better efficiency in general, with average runtime around 6.75s compared to 643s presented by the Su's approach. Such a difference is observed for circuits with more than ten inputs, where our method has a scalable temporal behavior, whereas the Su's approach presents a highly exponential trend.

5.2 Insights About the Runtime Reduction

In the presented work, the cube insertion procedure is based on the Su's approach and does not contain any specific speed-up techniques to accelerate it. Hence, the runtime

of the proposed method was expected to be comparable or even greater than the presented by Su when considering the computational impact of the cube removal procedure. In that way, the runtime reduction, shown in Section 5.1, is due to details on the implementation of both methods. As Su's approach implementation code is not available, it is hard to know which implementation decisions lead to the runtime optimization. It is improbable that only one implementation details lead to the runtime optimization, but probably multiple details together have done it. Thus, this section aims the give some insights into which implementation decisions may reduce the final runtime.

In general, it is used as much as possible vectors and non ordered maps to stores different sets of data in the proposed method. Vectors are used because they have a constant access time when the integer key is known and are stored contiguously in memory, which optimizes the cache use. Non ordered maps are applied to store a key and a value, with the key not necessarily being an integer position as vectors. This structure has a constant time to access and insert elements on average. In some cases, it is used ordered maps and sets, which present a $logn$ time access and insertion time, but allows the sorting of the map elements, which can be helpful in some procedures.

The implementation details that are the most probable impact factor for the runtime reduction is the cover data structure. As shown in Subsection 4.2.1, the cover data structure comprises two non ordered maps, one that relates the cubes with their respective COMs and the other that associate the minterms to the cubes that cover each of them. If the COMs and the minterms coverage have not been stored, it could be costly to runtime. For instance, to get the cubes that cover a minterm would be necessary to iterate over all cover cubes and to verify whether the minterm is covered by each one to get them. In the same way, to get a cube COMs, it would be needed to get all minterms covered by the cube, get all cover cubes that cover each minterm, and verify whose minterms are covered only one time to obtain them. In that way, the manner of storing the cover may increase the space complexity but may reduce the runtime complexity of two actions that are used multiple times during the approximation approach.

The Solutions structure is used to lessen the impact of the cover size by storing only the differences between the original cover and each partial solution. Even that the primary objective of using the Solutions data structure is to decrease the space complexity, the construction and storing of multiple and complete cover also impacts the runtime. Thus, the use of Solutions may be another reason to reduce the method runtime. It has been also intended to avoid passing the cover as a parameter to functions in the proposed

approach. As the cover is the biggest structure used, the main functions that use cover data are implemented inside the covers data structure, preventing copying the cover or memory indirections when passed as a parameter, so reducing the runtime impact.

The Su's initial approach uses Hasse diagrams to enumerate all possible cubes to generate the SCTs. As speed-up technique, Su proposes only considering the parents of each cube on the cover to generate the SCTs. If there are parent cubes with only one EIC, its parents are also considered. There are two differences between the Su's method and the proposed approach. The first one is that it has been verified that a search for the parent cubes of parent cubes with one EICs could increase the runtime but resulting in few optimizations on the final cover. As this case happens only in few circuits, it is disregarded in the proposed approach. The second difference is that the proposed work does not use the Hasse diagram concept, limiting the cubes enumeration to the cubes expansion. Depending on how the Hasse diagram is applied in Su's approach, it could increase the runtime.

It may have more reasons that lead to the runtime optimization, but as mentioned before, without the implementation code of Su's approach it is hard to be sure about which details have more or less impact on the associated runtime.

5.3 Results with Error Rate

Fixing an NoE threshold may be a problem because the ER depends on the number of input combinations related to the target circuit. For instance, the 16 NoE applied before corresponds to an ER of 12,5% for a circuit with 7 inputs but 0.0001% for a circuit with 17 inputs. As our method presents a good runtime efficiency, it is possible to apply a higher NoE and consequently allow the use of percentage error rate as the error metric. In Table 5.2 is shown the approximate cover results considering ER of 1%, 3% and 5% in benchmark circuits with more than 10 inputs. We have focused on these circuits to get NoE thresholds greater than 16. Column 1 presents the circuits and their input and output number. Column 2 and column 3 provide the ER in percentage and the corresponding NoE, respectively. Column 4 presents the number of literals for the respective circuit without approximations. Column 5 shows the literal count of the approximate covers obtained from our method and the literal reduction. Column 6 presents the number of cubes for the respective circuit without approximations. Column 7 shows the cubes count of the approximate covers obtained from our method and the cube reduction. Column 8

provides the runtime, whereas Column 9 provides the cube insertion procedure saturation.

Table 5.2 – Results from proposed method considering ER threshold in IWLS'93 benchmark suit.

Circuit	ER	NoE	Literals		Cubes		Time (s)	Saturation
			Original	Approximate	Original	Approximate		
sao2	1%	10	496	273 (0.55)	58	33 (0.56)	0.83	No
i: 10	3%	30		75 (0.15)		12 (0.20)	3.14	No
o: 4	5%	51		31 (0.06)		7 (0.12)	4.74	40
ex1010	1%	10	2718	2659 (0.97)	284	279 (0.98)	1.27	No
i: 10	3%	30		2588 (0.95)		273 (0.96)	3.92	No
o: 10	5%	51		2510 (0.92)		267 (0.94)	6.93	No
alu4	1%	163	5087	3732 (0.73)	575	461 (0.80)	148.22	135
i: 14	3%	491		2693 (0.52)		356 (0.62)	286.48	135
o: 8	5%	819		2139 (0.42)		297 (0.51)	429.81	135
misex3	1%	163	7784	6253 (0.80)	690	554 (0.80)	154.03	No
i: 14	3%	491		4796 (0.61)		436 (0.63)	217.58	199
o: 14	5%	819		3749 (0.48)		350 (0.50)	253.85	199
table3	1%	163	2644	1271 (0.48)	175	93 (0.53)	32.59	118
i: 14	3%	491		536 (0.20)		42 (0.24)	36.56	118
o: 14	5%	819		189 (0.07)		17 (0.10)	44.44	118
misex3c	1%	163	1561	499 (0.32)	197	89 (0.45)	133.55	54
i: 14	3%	491		469 (0.30)		85 (0.43)	229.03	54
o: 14	5%	819		447 (0.28)		82 (0.41)	342.40	54
b12	1%	372	207	193 (0.93)	43	40 (0.93)	2.34	0
i: 15	3%	983		167 (0.80)		35 (0.81)	7.52	0
o: 9	5%	1638		145 (0.70)		28 (0.65)	15.71	0
t481	1%	655	5233	1992 (0.38)	481	212 (0.44)	4.67	12
i: 16	3%	1966		942 (0.18)		120 (0.24)	6.00	12
o: 1	5%	3276		578 (0.11)		84 (0.17)	8.85	12
table5	1%	1310	2501	720 (0.28)	158	55 (0.34)	152.81	49
i: 17	3%	3932		278 (0.11)		24 (0.15)	302.28	49
o: 15	5%	6553		152 (0.06)		14 (0.08)	375.89	49

Our method reaches an average literal reduction of 37% with ER of 1%, 54% with ER of 3%, and 63% with an ER of 5%. For *table3* and *t481* circuits, we have obtained a literal count reduction close to 90% with an ER of 5%, and up to 94% with the same ER for *sao2* and *table5* circuits. The results of cube reduction tend to be similar to the literal reduction. Moreover, even though the *b12* could not be approximated with a 16 NoE, when the ER percentage is used as a constraint, it can be approximated with a literal count reduction up to 30%. Even with a higher NoE in circuits with many inputs, the runtime remains under 8 minutes.

The runtime draws attention because it is not correlated to the number of inputs/outputs or literals of the circuit. Besides the specific characteristics of each circuit, the saturation of the cube insertion procedure is another explanation. This saturation con-

sists in the NoE which the cube insertion procedure stops generating new partial solutions because it is not possible to obtain SCTs with one or two EICs. If the result is "No", the insertion method included all available NoE. In general, the NoE defines the number of times the cubes insertion and removal approaches are executed. As only the best two partial solutions are approximated, the number of executions of each approach is close to $2 * NoE$, which occurs in cases where there is no saturation. When there is an saturation value S that is less than the NoE threshold, the number of both approaches executions is up to $2 * S$. In the extreme case where S is equal to zero in *b12* circuit, each approach is applied only once. In that way, saturation has a substantial impact on the runtime of this method because it limits the number of executions of each approach.

5.4 Post-Processing Result

As shown in Section 4.6, it is possible to apply the Espresso tool to optimize the approximate cover and an ISBF version of that one. The approximate cover and the ISBF version are referred to as CSBF and ISBF for simplicity. This section presents the experimental results of using each optimization approach and some discussions about them. These experimental results are shown in Table 5.3, and have been obtained by applying both optimization approaches in the resulting approximate cover considering the same circuits and ERs taken into account in Table 5.2. The circuits name are in the first column in Table 6.1. Column 2 contains the ERs and the respective NoE. Columns 3, 4 and 5 present the number of literals, number of cubes and the final value of ER and NoE obtained using the Espresso over the CSBF, whereas column 6, 7 and 8 present the results obtained using the Espresso over the ISBF.

In Table 5.3, the executions for each optimization that result in fewer literals are highlighted in bold. ISBF optimization results in fewer literals than the CSBF optimization in more than half of executions, whereas the CSBF optimization leads to better results in only three executions. In cases where the CSBF leads to better results, the difference between the optimization results is only a few literals, with the most significant difference being nine literals. In contrast, when the ISBF optimization leads to better results, it presents a literal difference of up to 179 literals, which represents 26% of reduction concerning the CSBF optimization result.

The ER and NoE presented in Table 5.3 for CSBF optimization are the same as the approximation result, while the ISBF optimization can reduce these error metrics. The ER

Table 5.3 – Post Processing

Circuit	ER (NoE)	Espresso over CSBF			Espresso over ISBF		
		Literals	Cubes	ER (NoE)	Literals	Cubes	ER (NoE)
sao2	1% (10)	274	33	1.0% (10)	**273**	33	1.0% (10)
i = 10	3% (30)	79	33	3.0% (30)	**75**	**12**	**2.8% (29)**
o = 4	5% (51)	37	7	4.9% (50)	**21**	7	**3.6% (37)**
ex1010	1% (10)	2659	279	1.0% (10)	2659	279	1.0% (10)
i = 10	3% (30)	2588	273	3.0% (30)	2588	273	**2.7% (28)**
o = 10	5% (51)	2511	267	4.9% (50)	**2510**	267	4.9% (50)
alu4	1% (163)	**3732**	461	1.0% (163)	3741	461	1.0% (163)
i = 14	3% (491)	2693	356	3.0% (491)	2693	356	3.0% (491)
o = 8	5% (819)	2139	297	4.9% (798)	2139	297	4.9% (798)
misex3	1% (163)	**6253**	**554**	0.9% (159)	6258	555	**0.9% (153)**
i = 14	3% (491)	4978	459	2.9% (477)	**4796**	**436**	**2.8% (461)**
o = 14	5% (819)	3974	379	4.9% (796)	**3749**	**350**	**4.8% (788)**
table3	1% (163)	1286	93	0.9% (159)	**1271**	93	0.9% (159)
i = 14	3% (491)	536	42	2.9% (489)	536	42	2.9% (489)
o = 14	5% (819)	198	17	4.9% (809)	**189**	17	4.9% (809)
misex3c	1% (163)	678	112	0.8% (140)	**499**	**89**	**0.5% (84)**
i = 14	3% (491)	572	98	2.9% (474)	**469**	**85**	**1.5% (244)**
o = 14	5% (819)	514	90	4.7% (766)	**447**	**82**	**2.3% (376)**
b12	1% (372)	**193**	40	0.7% (256)	195	40	0.7% (256)
i = 15	3% (983)	171	36	2.9% (960)	**167**	**35**	2.9% (960)
o = 9	5% (1638)	154	33	4.9% (1600)	**145**	**28**	**4.7% (1536)**
t481	1% (655)	1992	212	0.9% (649)	1992	212	0.9% (649)
i = 16	3% (1966)	942	120	2.9% (1956)	942	120	2.9% (1956)
o = 1	5% (3276)	578	84	4.9% (3221)	578	84	4.9% (3221)
table5	1% (1310)	720	55	0.9% (1292)	720	55	0.9% (1292)
i = 17	3% (3932)	280	24	2.9% (3836)	**278**	24	2.9% (3836)
o = 15	5% (6553)	153	14	4.9% (6396)	**152**	14	4.9% (6396)

and NoE values highlighted in bold represent the execution where the ISBF optimization reduced the error inserted by the approximation. This error reduction is not the main objective of the optimization. However, it can be seen as a secondary optimization as fewer errors mean that the circuit is closer to the original behavior without increasing the literal count. The present work does not address approximate circuits already approximated, which could further reduce the number of literals in the cases where the error is reduced. In particular, the misex3c circuit presents a significant reduction in errors, inserting less than half of the allowed errors. In this case, if another approximation were made, it could be approximated with an NoE threshold of 443.

Hence, if it had to choose only one optimization approach to be used as post-processing, the ISBF approach would be more suitable because it leads to more literal reduction and can allow more approximation rounds by reducing the ER and NoE.

6 MULTILEVEL COMPARISON

This section presents a discussion about the impact of using two-level approximation tools to obtain an approximate multilevel circuits. Some of the existing multilevel papers propose the use of traditional two-level optimizations to approximate a multilevel circuit. Our objective in this section is not to analyze the use of two-level optimization inside a multilevel tool but to apply a complete two-level tool to approximate an two-level circuit, and then generate a final approximate multilevel circuit. We will refer a the two-level representation of the circuit as a 2L cover.

To represent the final multilevel circuit, we have adopted AIGs. There is a miscorrelation between optimizations realized in a 2L cover and a multilevel circuit based on this two-level description. For instance, optimizations on two-level circuit literals may not reduce node count in the AIG. The main reason for this miscorrelation is the differences between the characteristics of these two Boolean structures. While an 2L cover comprises a fixed number of levels and does not limit the number of literals in a cube (AND gates), an AIG does not limit the number of levels, and each AND gate must have only two inputs. The flexibility on levels allows optimizations in the number of AIG nodes by reusing logic structures and preventing redundant subcircuits. In another way, the use of AND2 represents, in a certain manner, the worst-case implementation of each cube.

Even with the mentioned miscorrelation between two-level and multilevel optimizations, it is interesting to investigate how significant the literals optimization is obtained through two-level approximation, as discussed in Section 5, when it is translated to AIG. It is also interesting to compare these AIGs with the ones approximated by applying only multilevel approximation tools and mixing these two approximations, using a two-level and a multilevel approximation tool together. To realize this experiment, it is applied the two-level approach presented in this work, called herein as 2LALS, and the state-of-the-art multilevel approximation tool ALSRAC (MENG; QIAN; MISHCHENKO, 2020). The ALSRAC are presented in Section 3.4 and we will give more details about this approach in the next.

The ALSRAC method approximates an AIG considering ER and other two magnitude metrics as the error constraint, but we are taking into account only the ER constraint in this work. The ALSRAC comprises of two main phases.

The first one enumerates to each AIG node *n* the other nodes i_a that may substitute one input of *n* without modifying the AIG logic behavior considering N simulation

rounds, without changing the other input i_o of *n*. The nodes i_a could be any node in the AIG, but for scalability reasons, ALSRAC only considers the nodes in paths between *n* and the primary inputs. The traditional logic synthesis also uses substitution techniques but for that it is applied a formal method, like SAT, to guarantee the logic equivalence of circuits before and after substitutions. Since it is an ALS method, the logical equivalence verification through simulations is enough.

The second phase consists of obtaining a subcircuit for each possible substitution enumerated where the subcircuit output and inputs are the output of node *n* and the inputs of i_o and i_a. Then a truth table is constructed by considering the input/output relationship observed by simulating the circuit. If a given input combination does not happen during simulation, it is considered a *don't care*. At the end, an optimized cover for this truth table is obtained using the Espresso tool, and then it is converted to an AIG, replacing the original subcircuit.

These two phases are executed iteratively, choosing the subcircuit replacement that leads to a more significant node reduction that does not exceed the ER threshold until this ER limit is reached. It is worth mentioning that the approximations performed during the ALSRAC execution are dependent on the simulation vectors selected. As these vectors are selected randomly, so this method can result in different approximations for the same circuit and ER threshold.

The approximate AIG is optimized using the ABC tool. For that, we execute the *resyn2* script iteratively until no more gains in the number of nodes or levels are obtained for ten consecutive iterations. The conversion from the 2L cover to AIG is also done using ABC.

To analyze the impact of two-level approximation in multilevel circuits and the use of 2LALS and ALSRAC together, we have distributed the ER threshold between these approximation tools. The ER allowed in 2LALS is called ER_{2L}, and in ALSRAC is ER_{ML}, with both values being integers between 0 and ER. In that way, $ER = ER_{2L} + ER_{ML}$. The flow 1 regarding this analysis has the following sequence:

1. The initial 2L cover is approximated with 2LALS with ER threshold of ER_{2L}.
2. The approximated 2L cover is converted to AIG using the ABC tool.
3. The ALSRAC approximates the AIG with ER threshold of ER_{ML}.
4. The approximated AIG is optimized with ABC.
5. Converts the final AIG to a 2L cover using ABC.

6. Optimize the 2L cover with Espresso.

The cover obtained at the end of the process described in flow 1 is used to get data regarding the number of literals, which will be compared to the results obtained with flow 2. Although the objective of this investigation is to analyze the resulting multilevel circuit, the data about literals count was accessible and may help the understanding of some behaviors and fits into the general context of this work. The flow 2 that inverts the sequence of approximation tools is presented in following:

1. Convert the original 2L cover in AIG using ABC.
2. The ALSRAC approximates the AIG with ER threshold of ER_{ML}.
3. The approximated AIG is optimized with ABC.
4. Converts the optimized AIG to 2L cover using ABC.
5. Optimize the 2L cover with Espresso.
6. The 2L cover is approximated with 2LALS with ER threshold of ER_{2L}.
7. The final 2L cover is converted into a final AIG using the ABC tool.

To get an reasonable amount of data the ER threshold is fixed in 5% with the following distribution of ER_{2L} and ER_{ML}: (5% - 0%), (4% - 1%), (3% - 2%), (2% - 3%), (1% - 4%) and (0% - 5%). Note that the distribution (5% - 0%) implies only approximate the circuit with 2LALS and (0% - 5%) in only with ALSRAC. At the end of each flow, we compare the truth table of the approximate circuit and the original circuit to guarantee that the ER of 5% threshold is satisfied.

In Table 6.1 is presented the data obtained with both flows. The circuits used in this experiment are the same as used in Table 5.2 with their name in the first column in Table 6.1. The first column also contains the partial ER ER_{2L} and ER_{ML}, respectively. In columns 2, 3 and 4, are the cover literals, and the number of nodes and levels of the AIG are obtained by flow 1. With the same sequence in columns 5, 6, and 7 are the data about flow 2. In the lines containing the circuit name are its original values of the number of literals, nodes and levels. The numbers in bold represents the best solution for the metric for the associate circuit.

Looking at Flow 1 AIG results, it is clear that does not exist a correlation between the ER distribution and the number of nodes and levels, and it is possible to note three distinct behaviors:

1. The best results are obtained when only ALSRAC is applied to approximate the

Table 6.1 – Two-Level approximation impact on multilevel circuits

Circuit	Flow 1			Flow 2		
ER_{2L}-ER_{ML}	literal	and	level	literal	and	level
sao2	496	134	10	496	134	10
5% - 0%	49	15	5	37	15	5
4% - 1%	34	14	5	41	16	6
3% - 2%	37	16	6	58	26	6
2% - 3%	67	27	7	52	23	6
1% - 4%	88	35	7	**5***	**1***	**1***
0% - 5%	85	30	8	85	29	8
ex1010	2718	1895	14	2718	1895	14
5% - 0%	6706*	**1735**	14	**2511**	1735	14
4% - 1%	2614	**1735**	14	2568	1794	14
3% - 2%	2694	1744	**13**	2624	1820	14
2% - 3%	2695	1762	14	2707	1830	14
1% - 4%	2777*	1765	14	2775	1847	14
0% - 5%	2912*	1774	14	2912*	1848	14
alu4	5087	1138	19	5087	1138	19
5% - 0%	2239	611	**13**	**2139**	611	13
4% - 1%	2290	578	14	2247	585	14
3% - 2%	2428	604	15	2267	597	14
2% - 3%	2534	535	15	2450	692	16
1% - 4%	2742	472	14	2750	671	14
0% - 5%	3042	**437**	15	3042	643	16
misex3	7784	1955	20	7784	1955	20
5% - 0%	6599	1268	17	**3974**	1268	17
4% - 1%	4553	1111	19	4186	1355	20
3% - 2%	4971	1010	19	4470	1349	19
2% - 3%	5395	883	19	4838	1389	19
1% - 4%	5789	686	19	5587	1549	20
0% - 5%	6217	**473**	**14**	6217	1511	19
table3	2644	1511	20	2644	1511	20
5% - 0%	470	**155**	11	**198**	155	11
4% - 1%	261	174	12	246	179	12
3% - 2%	313	190	13	360	253	12
2% - 3%	299	178	12	364	257	13
1% - 4%	270	164	**9**	381	253	12
0% - 5%	399	183	10	399	203	11
misex3c	1561	553	18	1561	553	18
5% - 0%	641	254	11	514	254	11
4% - 1%	491	243	11	537	268	11
3% - 2%	523	254	**10**	488	242	**10**
2% - 3%	523	253	**10**	**472**	**233**	**10**
1% - 4%	554	261	11	508	246	10
0% - 5%	577	260	11	577	269	11
b12	207	54	6	207	54	6
5% - 0%	165	45	**5**	**153**	45	5
4% - 1%	165	49	7	170	55	7
3% - 2%	171	52	7	160	52	7
2% - 3%	156	45	7	164	48	7
1% - 4%	154	42	6	189	49	6
0% - 5%	165	**41**	6	165	43	6
t481	5233	52	10	5233	52	10
5% - 0%	**578**	69*	9	**578**	69	9
4% - 1%	588	48	7	631	69	10
3% - 2%	601	36	**6**	799	57	9
2% - 3%	613	33	**6**	793	69	10
1% - 4%	1217	43	**6**	1607	74	10
0% - 5%	2113	**24**	7	2113	24	7
table5	2501	1326	21	2501	1326	21
5% - 0%	412	86	**8**	**153**	86	8
4% - 1%	170	**70**	9	196	89	10
3% - 2%	177	76	9	200	107	10
2% - 3%	195	89	9	264	159	12
1% - 4%	248	101	11	275	159	12
0% - 5%	290	120	10	290	129	10

circuit, as in misex3. In that case, the optimizations performed in 2LALS do not lead to optimizations on the AIG. The best number of level in t481 are obtained in when ER_{ML} is 4, but it also fits on this behavior.

2. The optimizations performed by running 2LALS and ALSRAC together lead to the best AIG, as in sao2, misex3c, table5. In this case, the optimizations realized by 2LALS are not undone when the 2L cover is converted in AIG, working together with optimizations made by ALSRAC to optimize the final circuit.
3. The last behavior consists of the best number of nodes appearing with high ER_{2L} and the best number of levels in high ER_{ML}, and vice-versa, as in alu4, table3 e b12. This one is hard to figure out the impact of each ALS tool, requiring further investigation considering the characteristics of each circuit.

It is worth to mention that the gain during the executions is dependent on the circuit characteristics, which explain these different behaviors. When comparing both flows related to AIG data, in 7 of 9 circuits, the best results in nodes and levels are obtained according to the flow 1, showing that applying 2LALS initially and then using the ALSRAC to approximate the AIG leads to the best results. Similarly, in only one circuit the best literals count has resulted from Flow 1. Moreover, in 7 of 9 circuits, the best literals count obtained by adopting the flow 2 has been resulted from executing only 2LALS, showing that the approximation on multilevel does not impact the final 2L cover.

Three cases catch out attention because they do not fit on other execution behaviors. These cases are marked with an asterisk and will be discussed below.

The first case happens in ex1010 circuit solutions. The marked numbers of literals are higher than the original number of literals. The (5% - 0%) execution in particular results is more than double the number of literals in the original circuit. Note that the approximation is only made by applying 2LALS. In that way, this worsening may be caused by the conversion to AIG, optimizing it, and returning to a 2L cover. This behavior exemplifies a miscorrelation between the optimizations in AIG and final 2L cover. Moreover, in this circuit, the results from flow 1 related to AIG metrics do not fit the three groups discussed before. The better results happen with high ER_{2L}, which may mean that the approximations in the 2L cover have a better impact than those done in the AIG. We have not mentioned this behavior before because the difference between the best and the worst number of nodes is about 2%, difficulting a more detailed analysis.

The second case is about flow 1 in t481 circuit with (5% - 0%) ER distribution. The approximates AIG contains more nodes (69) than the original AIG (52). As 2LALS only

make the approximation, this worsening happens because the 2L cover approximations limit the optimizations on the AIG. This case is an example of the miscorrelation between optimization on the 2L cover and final AIG.

The last case is about the results from flow 2 related to sao2 circuit approximation with (1% - 4%) ER distribution. These results have shown a huge reduction in both 2L cover and AIG metrics. Such a reduction is contrary to two trends: the first one looking to only sao2 results, where the greater is the ER_{ML} the worst is the quality of the approximated circuit, and the second one, as discussed before, the best results for the number of literals happens with high ER_{ML}. We have made a more detailed analysis in this circuit to understand why such a reduction has been obtained. We have concluded that the significant optimizations are done with ALSRAC with 4% of ER_{ML}. It is expected that when the ER_{ML} is 5%, the result would be at least equal to the one with 4%. However, as ALSRAC presents a stochastic behavior, this optimization could not be seen during our execution but may be possible. Due to this stochastic behavior, it is hard for us to understand which approximations have been made to optimize the circuit in this execution more than others. Another interesting point about this execution is that both 2L cover and AIG are probably the exact solution for sao2 with 5% ER. This exactness shows how far other executions, even the results for 2LALS in Table 5.2, can be from the best possible approximation. Probably, the other approximated circuits are also far from the exact solution.

With the presented experiment, it is possible to state three main conclusions. When the objective is an approximate multilevel circuit, using two-level and multilevel ALS methods to approximate multilevel circuits leads to better results than by applying a multilevel approximation method itself in most of the considered circuits, which suggests that the use of 2L-ALS methods can result in better approximate multilevel circuits. When the objective is an approximate two-level circuit, a multilevel ALS method does not lead to a better approximate 2L cover than a 2L-ALS method alone, showing that the use of multilevel ALS is not indicated to generate two-level approximate circuits. The procedure described in flow 1 generates the best AIG results, whereas the one from flow 2 has provided the best 2L cover results, suggesting that approximating the aimed structure in the final may leads to improvement.

7 CONCLUSION

With the growing use of error-resilient applications, it is interesting to design circuits that take advantage of this application characteristic to optimize its cost and performance. In that way, the development of tools to synthesize approximate circuits that generate a more optimized circuit in a scalable way is aimed.

In this work, we proposed the first two-level approximate logic synthesis method that exploits both insertion and removal of cubes to approximate a cover considering ER. Experimental results have shown that our method surpasses the previous state-of-the-art method in quality of results and scalability. Our source code, the applied benchmark circuits to generate the experimental results, and their approximate descriptions are publicly available on GitHub (AMMES, 2020).

We also presented an investigation about using two-level and multilevel ALS approaches together. This investigation leads to three main conclusions.

- When the objective is an approximate multilevel circuit, using two-level and multilevel ALS methods to approximate multilevel circuits tends to provide better results than applying just a multilevel approximation approach.
- When the objective is to approximate two-level circuits, using just a 2L-ALS approach leads to better approximate 2L cover. It shows that the use of multilevel ALS is not indicated to generate two-level approximate circuits.
- Approximating the target structure in the final of the execution flow leads to better solutions, e.g., if the target structure is a multilevel circuit, using an ML-ALS tool in the final is the better choice.

We want to expand this work in two directions as future works:

- Modify the proposed approach to approximate functions using different magnitude error metrics as the error constraint. The use of magnitude error metrics is interesting to allow a more detailed approximation of arithmetic circuit designs.
- Apply the proposed method as the local approximation approach of a multilevel ALS approach. Some works use traditional two-level techniques to assist the local approximation of a multilevel circuit. The use of an ALS approach in subcircuits of a multilevel circuit can be relevant.

REFERENCES

AKERS, S. Binary decision diagrams. **IEEE Transactions on computers**, IEEE, C-27, n. 6, p. 509–516, 1978.

AMANT, R. S. et al. General-purpose code acceleration with limited-precision analog. In: **2014 International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2014.

AMMES, G. **A Two-Level Approximate Logic Synthesis Combining Cube Insertion and Removal**. 2020. Available from Internet: <https://github.com/GabrielAmmes/2LALS-IR>.

BANERES, D.; CORTADELLA, J.; KISHINEVSKY, M. A recursive paradigm to solve boolean relations. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 58, n. 4, p. 512–527, 2019.

BHARDWAJ, K.; MANE, P. S.; HENKEL, J. Power- and area-efficient approximate wallace tree multiplier for error-resilient systems. In: **International Symposium on Quality Electronic Design**. [S.l.: s.n.], 2014.

BORNHOLT, J.; MYTKOWICZ, T.; MCKINLEY, K. S. Uncertain<t>: a first-order type for uncertain data. **ACM SIGARCH Computer Architecture News**, ACM, v. 42, n. 1, 2014.

BRAND, D. Verification of large synthesized designs. In: **1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 1993.

BRAYTON, R.; HACHTEL, G.; SANGIOVANNI-VINCENTELLI, A. Multilevel logic synthesis. **Proceedings of the IEEE**, IEEE, v. 78, n. 2, p. 264–300, 1990.

BRAYTON, R.; SOMENZI, F. An exact minimizer for Boolean relations. In: **1989 International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 1989.

BRAYTON, R. K. et al. **Logic Minimization Algorithms for VLSI Synthesis**. [S.l.]: Kluwer Academic Publishers, 1984.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, IEEE, v. 100, n. 8, p. 677–691, 1986.

CHANDRASEKHARAN, A. et al. Precise error determination of approximated components in sequential circuits with model checking. In: **2016 Design Automation Conference (DAC)**. [S.l.: s.n.], 2016.

CHANDRASEKHARANA, A. et al. Approximation-aware rewriting of AIGs for error tolerant applications. In: **2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2016.

CHEN, L. et al. On the design of approximate restoring dividers for error-tolerant applications. **IEEE Transactions on Computers**, IEEE, v. 65, n. 8, p. 2522–2533, 2016.

CHEN, W.-K. **The VLSI Handbook**. [S.l.]: CRC Press, 2006.

CHIPPA, V. et al. Analysis and characterization of inherent application resilience for approximate computing. In: **2013 Design Automation Conference (DAC)**. [S.l.: s.n.], 2013.

CHO, K. et al. edram-based tiered-reliability memory with applications to low-power frame buffers. In: **2014 International Symposium on Low Power Electronics and Design (ISLPED)**. [S.l.: s.n.], 2014.

ESMAEILZADEH, H. et al. Neural acceleration for general-purpose approximate programs. In: **2012 Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2012.

ESMAEILZADEH, H. et al. Quality programmable vector processors for approximate computing. In: **2012 Architectural Support for Programming Languages and Operating Systems (ASPLOS)**. [S.l.: s.n.], 2012.

EéN, N.; SöRENSSON, N. An extensible sat-solver. **Giunchiglia E., Tacchella A. (eds) Theory and Applications of Satisfiability Testing**, Springer, Berlin, Heidelberg, v. 2919, p. 502–518, 2004.

FROEHLICH, S.; GROßE, D.; DRECHSLER, R. Approximate hardware generation using symbolic computer algebra employing gröbner basis. In: **2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2018.

FROEHLICH, S.; GROßE, D.; DRECHSLER, R. One Method - All Error-Metrics: A Three-Stage Approach for Error-Metric Evaluation in Approximate Computing. In: **2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2019.

FROEHLICH, S.; GROßE, D.; DRECHSLER, R. One Method - All Error-Metrics: A Three-Stage Approach for Error-Metric Evaluation in Approximate Computing. In: **2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2019.

GORDON, A. D. et al. Probabilistic programming. In: **2014 Future of Software Engineering (FOSE) Proceedings**. [S.l.: s.n.], 2014.

HACHTEL, G. D.; SOMENZI, F. **Logic synthesis and verification algorithms**. [S.l.]: Springer Science & Business Media, 2006.

HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. In: **Proc. 18th IEEE Eur. Test Symp. (ETS)**. [S.l.: s.n.], 2013.

JIANG, H.; HAN, J.; LOMBARDI, F. A Comparative Review and Evaluation of Approximate Adders. In: **Great Lakes Symposium on VLSI (GLSVLSI)**. [S.l.: s.n.], 2015.

JIANG, H. et al. A comparative evaluation of approximate multipliers. In: **2016 International Symposium on Nanoscale Architectures (NANOARCH)**. [S.l.: s.n.], 2016.

KARNAUGH, M. The map method for synthesis of combinational logic circuits. **IEEE Transactions of the American Institute of Electrical Engineers**, IEEE, v. 72, n. 5, p. 593–599, 1953.

KARPUZCU, U.; AKTURK, I.; KIM, N. S. Accordion: Toward soft near-threshold voltage computing. In: **2014 International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2014.

KUBICA, M.; KANIA, D. Graph of outputs in the process of synthesis directed at cplds. **Mathematics**, v. 7, p. 1171, 12 2019.

KUEHLMANN, A.; KROHM, F. Equivalence checking using cuts and heaps. In: **1997 Design Automation Conference (DAC)**. [S.l.: s.n.], 1997.

KULKARNI, P.; GUPTA, P.; ERCEGOVAC, M. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In: **2011 International Conference on VLSI Design**. [S.l.: s.n.], 2011.

LIU, G.; ZHANG, Z. Statistically certified approximate logic synthesis. In: **2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2017.

LIU, W. et al. Inexact floating-point adder for dynamic image processing. In: **International Conference on Nanotechnology**. [S.l.: s.n.], 2014.

MAHDIANI, H. R. et al. Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. **IEEE Transactions on Circuits and Systems**, v. 57, n. 4, p. 850 – 862, 2010.

MCCLUSKEY, E. J. Minimization of boolean functions. **The Bell System Technical Journal**, v. 35, n. 6, p. 1417–1444, 1956.

MCELVAIN, K. **IWLS93 Benchmark Set: Version 4.0.** 1993. Available from Internet: <https://ddd.fit.cvut.cz/prj/Benchmarks/IWLS93.pdf>.

MENG, C.; QIAN, W.; MISHCHENKO, A. ALSRAC: Approximate Logic Synthesis by Resubstitution with Approximate Care Set. In: **2020 Design Automation Conference (DAC)**. [S.l.: s.n.], 2020.

MIAO, J.; GERSTLAUER, A.; ORSHANSKY, M. Approximate logic synthesis under general error magnitude and frequency constraints. In: **2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2013.

MIAO, J.; GERSTLAUER, A.; ORSHANSKY, M. Multi-level approximate logic synthesis under general error constraints. In: **2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2014.

MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MISAILOVIC, S. et al. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In: **2014 International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)**. [S.l.: s.n.], 2014.

MISAILOVIC, S. et al. Quality of service profiling. In: **2010 International Conference on Software Engineering**. [S.l.: s.n.], 2010.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Improvements to technology mapping for lut-based fpgas. In: **2006 International symposium on Field programmable gate arrays (FPGA)**. [S.l.: s.n.], 2006.

MITTAL, S. A survey of techniques for approximate computing. **ACM Computing Surveys**, ACM, v. 48, n. 4, 2016.

MOHAPATRA, D. et al. Design of voltage-scalable meta-functions for approximate computing. In: **2011 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2011.

MRAZEK, V. et al. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: **2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2017.

PAN, P.; LIN, C.-C. A new retiming-based technology mapping algorithm for lut-based fpgas. In: **1998 International symposium on Field programmable gate arrays (FPGA)**. [S.l.: s.n.], 1998.

PARHAMI, B. **Computer Arithmetic: Algorithms and Hardware Designs**. [S.l.]: Oxford University Press, 2009.

RABAEY, J.; CHANDRAKASAN, A.; NIKOLIC, B. **Digital integrated circuits.** [S.l.]: Prentice hall Englewood Cliffs, 2002.

RANJAN, A. et al. ASLAN: Synthesis of approximate sequential circuits. In: **2014 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2014.

REHMAN, S. et al. Architectural-space exploration of approximate multipliers. In: **2016 International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2016.

SAADAT, H.; JAVAID, H.; PARAMESWARAN, S. Approximate integer and floating-point dividers with near-zero error bias. In: **2019 Design Automation Conference (DAC)**. [S.l.: s.n.], 2019.

SAMPSON, A. et al. Enerj: approximate data types for safe and general low-power computation. In: **2011 Conference on Programming Language Design and Implementation (PLDI)**. [S.l.: s.n.], 2011.

SAMPSON, A. et al. Approximate storage in solid-state memories. In: **2013 Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2013.

SCARABOTTOLO, I. et al. Approximate logic synthesis: A survey. **Proceedings of the IEEE**, v. 108, n. 12, p. 2195–2213, 2020.

SHIN, D.; GUPTA, S. K. Approximate logic synthesis for error tolerant applications. In: **2010 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2010.

SHIN, D.; GUPTA, S. K. A new circuit simplification method for error tolerant applications. In: **2011 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2011.

SIDIROGLOU-DOUSKOS, S. et al. Managing performance vs. accuracy trade-offs with loop perforation. In: **2011 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2011.

SOEKEN, M. et al. BDD minimization for approximate computing. In: **2016 Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2016.

SU, S.; WU, Y.; QIAN, W. Efficient Batch Statistical Error Estimation for Iterative Multi-level Approximate Logic Synthesis. In: **2018 Design Automation Conference (DAC)**. [S.l.: s.n.], 2018.

SU, S. et al. A novel heuristic search method for two-level approximate logic synthesis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 3, p. 654–669, 2020.

TSEITIN, G. S. **On the complexity of derivation in propositional calculus**. [S.l.]: Springer Berlin Heidelberg, 1983.

UMANS, C.; VILLA, T.; SANGIOVANNI-VINCENTELLI, A. Complexity of two-level logic minimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 25, n. 7, p. 1230–1246, 2006.

VASICEK, Z. Relaxed equivalence checking: a new challenge in logic synthesis. In: **2017 IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)**. [S.l.: s.n.], 2017.

VASICEK, Z. Formal methods for exact analysis of approximate circuits. **IEEE Access**, IEEE, v. 7, p. 177309 – 177331, 2019.

VASICEK, Z.; SEKANINA, L. Evolutionary design of complex approximate combinational circuits. **Genetic Programming and Evolvable Machines**, Springer, v. 17, p. 169–192, 2016.

VENKATARAMANI, S. et al. Quality programmable vector processors for approximate computing. In: **2013 International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2013.

VENKATARAMANI, S.; ROY, K.; RAGHUNATHAN, A. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In: **2013 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2013.

VENKATARAMANI, S. et al. SALSA: Systematic logic synthesis of approximate circuits. In: **2012 Design Automation Conference (DAC)**. [S.l.: s.n.], 2012.

WENDLER, A.; KESZOCZE, O. A fast BDD Minimization Framework for Approximate Computing. In: **2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)**. [S.l.: s.n.], 2020.

WESTE, N.; HARRIS, D. **Cmos vlsi design. A circuits and systems perspective**. [S.l.]: Addison-Wesley Publishing Company, 2010.

WU, Y.; QIAN, W. An efficient method for multi-level approximate logic synthesis under error rate constraint. In: **2016 Design Automation Conference (DAC)**. [S.l.: s.n.], 2016.

WU, Y.; QIAN, W. Alfans: Multilevel approximate logic synthesis framework by approximate node simplification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 7, p. 1470 – 1483, 2020.

WU, Y. et al. Approximate logic synthesis for FPGA by wire removal and local function change. In: **2017 Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2017.

XU, Q.; MYTKOWICZ, T.; KIM, N. S. Approximate computing: A survey. **IEEE Design & Test**, IEEE, v. 33, n. 1, p. 8–22, 2016.

XU, X.; HUANG, H. H. Exploring data-level error tolerance in high-performance solid-state. **IEEE Transactions on Reliability**, IEEE, v. 64, n. 1, p. 15–30, 2015.

YAO, Y. et al. Approximate Disjoint Bi-Decomposition and Its Application to Approximate Logic Synthesis. In: **2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2017.

ZHOU, Z. et al. DALS: Delay-driven Approximate Logic Synthesis. In: **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2018.

ZOU, C.; QIAN, W.; HAN, J. DPALS: A dynamic programming-based algorithm for two-level approximate logic synthesis. In: **2015 IEEE 11th International Conference on ASIC (ASICON)**. [S.l.: s.n.], 2015.

ČEšKA, M. et al. Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-bit Multipliers Accomplished. In: **2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2017.

APPENDIX A — RESUMO DA DISSERTAÇÃO

Método de Síntese Lógica Aproximada Dois-Níveis Baseado na Inserção e Remoção de Cubos.

A.1 Introdução

Computação aproximada é um paradigma que permite que um sistema tenha uma execução imprecisa ou inexata com o objetivo de otimizar o seu desempenho e sua eficiência energética. Quando este paradigma é aplicado em sistemas que executam funções resilientes a erros, é possível otimizar o sistema sem degradar de forma crítica a operação desejada. Este trabalho foca no uso de computação aproximada no nível de circuitos, em particular, nos circuitos integrados digitais.

Ferramentas computacionais fornecem um fluxo altamente automatizado para o desenvolvimento de projetos de circuitos integrados (CIs). Este fluxo pode ser dividido em três passos principais: síntese de alto nível, síntese lógica e síntese física. A síntese lógica, em particular, tem como objetivo otimizar a lógica do circuito e implementá-lo em uma dada tecnologia alvo. Esta etapa do fluxo de projeto é executada sobre representações dois-níveis ou multinível que implementam os blocos combinacionais de um dado circuito.

A aplicação de computação aproximada no nível de circuitos consiste em obter uma implementação que não é logicamente equivalente à especificação mas consegue realizar otimizações em área, desempenho e consumo de energia. Diversos trabalhos propõem técnicas para aproximar um circuito de forma automática através de modificação sistemática do funcionamento de um circuito genérico sem exceder uma dada restrição de erro. Devido à similaridade em técnicas, estruturas de dados e objetivos de otimização com a etapa de síntese lógica, a geração automática de circuitos aproximados é frequentemente chamada de síntese lógica aproximada.

A.1.1 Motivação e Proposta

Métodos de síntese lógica tradicional para construção de circuitos dois-níveis podem ser utilizados para síntese de componentes programáveis CPLDs, bem como parte

de métodos para síntese de circuitos multinível. Além da geração de expressões aproximadas do tipo soma-de-produtos (SOP) e produto-de-somas (POS), técnicas de aproximação para circuitos dois-níveis poder ser exploradas nesses dois cenários. Além disso, o entendimento dos conceitos e técnicas relacionados à aproximação dois-níveis pode contribuir significativamente para futuros estudos sobre a aproximação de circuitos multinível.

Este trabalho propõe um método para aproximar circuitos dois-níveis que tem como entrada uma SOP e um dado limite de frequência de erro, e gera uma expressão aproximada com um número de literais reduzido e que respeita o dado limite de erro. O método proposto foi desenvolvido com a intenção de ser escalável em relação à quantidade de erros permitidos, possibilitando uma inserção de mais erros do que é observado em outros trabalhos que abordam o mesmo problema.

A.2 Conceitos Preliminares

Uma breve revisão de fundamentos de síntese lógica aproximada e de métricas de erro é apresentada a seguir, bem como a terminologia adotada, buscando facilitar o entendimento do método de síntese lógica aproximada dois-níveis proposto.

Uma *variável* corresponde ao símbolo utilizado para representar sinais de entrada e saída. A ocorrência de uma variável é chamada de *literal*, podendo ser um literal de entrada ou de saída, depende da respectiva variável. Um literal de entrada pode ser direto ou complementado (negado). Um produto de literais onde cada variável aparece no máxima uma vez é chamado de *cubo*. O caso particular onde um cubo contém um literal para cada literal de entrada e somente um literal de saída é chamado de *mintermo*. Um conjunto de cubos que cobre todos os mintermos é chamado de *cobertura*. O caso especial de cobertura onde todos os cubos são referentes a mesma saída, ou seja, uma cobertura de saída única, pode ser representado por uma *soma-de-produtos* (SOP). Em geral, quando se representa a cobertura com uma SOP os literais de saída são omitidos. O tamanho de um cubo é igual ao número de mintermos cobertos por ele. A expansão de um cubo consiste em remover um de seus literais de entrada, tornando-o em um cubo maior.

No contexto da aproximação de circuitos, múltiplas métricas foram propostas com o objetivo de quantificar e controlar a quantidade de erro inserido (FROEHLICH; GROßE; DRECHSLER, 2019b). Neste trabalho utilizamos a métrica de *Error Rate* (ER) para limitar a ocorrência de erros. A métrica ER consiste na frequência em que os erros ocorrem ou

na probabilidade de um erro ocorrer para um dada combinação de entrada. Desta forma, esta métrica é calculada considerando a razão entre o número combinações de entrada que levam a erros e o número total de possíveis combinações de entrada.

Considerando a síntese lógica aproximada, uma combinação de entrada que leva a uma ou mais saídas incorretas é definida como uma combinação de entrada errônea (EIC, do inglês *erroneous input combination*). Por exemplo, se dois mintermos levam a saídas incorretas mas contém os mesmo literais de entrada, somente um EIC será levado em conta. Além disso, o número de erros (NE) em uma dada cobertura é igual aos seu número de EICs. Quando o NE é utilizado como métrica de erro, o seu valor pode ser tanto arbitrário quanto relativo ao número de entradas do circuito. Para o segundo caso, o NE é igual a $2^n * er$, onde n é o número de entradas e er o limite de ER.

A.3 Estado-da-Arte

Em (SU et al., 2020), Su *et al.* apresentam um método de busca heurística para resolver o problema de síntese lógica aproximada dois-níveis (2L-ALS) usando ER como métrica de erro. Este trabalho pode ser considerado o estado-da-arte para este problema e é apresentado a seguir.

O principal objetivo do método de Su é identificar um conjunto de combinações de entrada para terem sua entrada complementada de 0 para 1 que maximize a redução de literais da cobertura. Este conjunto de combinações é chamada de SICC e é equivalente a um conjunto de EICs. Eles propõe neste trabalho a estrutura de dados *SICC-cube tree* (SCT) que agrupa um conjunto de EICs com os cubos dependem destes EICs para serem inseridos na cobertura. Uma SCT consiste em uma árvore de dois-níveis, onde a raiz contém o conjunto de EICs e as folhas contém os cubos a serem inseridos na cobertura. O número de EICs na raiz é igual ao número de erros inseridos na cobertura.

Um cubo deve satisfazer duas condições para ser inserido como folha de uma SCT. Inicialmente, pelo menos um cubo deve ser removido da cobertura quando este é adicionado na cobertura. Por fim, o número de literais do cubo removido deve ser maior do que do cubo a ser inserido. Estas condição são impostas para garantir que os cubos nas folhas da SCT levem a otimizações no número de literais.

Inicialmente, todos os possíveis cubos de uma dada função utilizando um diagrama de Hasse. Esses cubos são utilizados para construir as SCTs. Em seguida, as SCTs obtidas são combinadas, pois existem SCTs com menos erros que o limite estabelecido.

Por fim, a SCT que leva a maior redução de literais é escolhida.

O procedimento para estimar a redução de literais de uma dada SCT contém três passos principais. Como a inserção de um cubo não garante que outros sejam removidos, o primeiro passo consiste em identificar quais cubos da cobertura podem ser removidos quando todos os cubos folhas de uma SCT são inseridos na cobertura. Além disso, inserir todos os cubos folhas pode aumentar o número de literais. Desta forma, o segundo passo é identificar quais cubos folhas são necessários para remover os cubos identificados no primeiro passo. Por fim, a redução de literais é calculada subtraindo o número de literais nos cubos removidos e nos cubos inseridos.

Além deste método base, os autores apresentam quatro técnicas de aceleração para permitir o uso em circuitos maiores.

1. Como a complexidade temporal do algoritmo-base cresce exponencialmente com o número de erros, o número de erros permitidos para cada execução é limitado em dois, gerando assim coberturas parcialmente aproximadas. Todas estas soluções parciais geradas são aproximadas iterativamente até que o erro máximo permitido seja atingido.
2. Com a utilização da primeira técnica, é gerada uma quantidade muito grande de soluções parciais é criado, tornando custosa a aproximação de todas estas soluções. Desta forma, somente as duas soluções parciais com a maior redução de literais para um dado número de erros são aproximadas.
3. Como o número de erros permitidos para cada aproximação é dois, é necessário combinar as SCTs que inserem somente um erro. Para diminuir o número de SCTs combinadas, somente um subconjunto de todas as SCTs são utilizadas na etapa de combinação. Para isso, inicialmente são estimados os literais das SCTs geradas sem combinação. Em seguida, duas SCTs são combinadas caso a primeira esteja entre as 25% melhores e a segunda entre as 80% melhores SCTs.
4. Considerar todos os cubos em um diagrama de Hasse implica em um alto custo computacional. Para reduzir este custo, eles consideram somente os cubos do diagrama de Hasse que são parentes dos cubos presentes na cobertura. Estes cubos podem ser vistos como os cubos obtidos removendo literais de entrada ou inserindo literais de saídas nos cubos da cobertura.

Os valores utilizados na três primeiras técnicas de aceleração foram obtidas utilizando análises empíricas. Mesmo com a utilização destas técnicas, o aumento no número

de erros permitidos continua sendo um desafio importante no tempo de execução desta técnica.

A.4 Trabalho Proposto

Em (SHIN; GUPTA, 2010), os autores apresentam duas técnicas para a aproximação de SOPs: a inserção de cubos na SOP, invertendo as saídas de mintermos de 0 para 1, e a remoção de cubos da SOP, modificando estas saídas de 1 para 0. Eles também propõem um experimento para comparar estas duas técnicas e sugerem que a inserção de cubos leva a resultados melhores do que a sua remoção. Baseado nessa conclusão, trabalhos relacionados subsequentes têm focado prioritariamente na inserção de cubos. A seguir, apresentamos um exemplo de aproximação utilizando estas duas técnicas individualmente e em conjunto.

Para a SOP $x_1 * \overline{x_2} + x_0 * x_2$ representada pelo mapa de Karnaugh (diagrama de Veitch) ilustrado na Figura A.1a, suas aproximações considerando um limite de número de erros igual a dois através da inserção de cubos, da remoção de cubos e de ambas em conjunto, são mostradas na Figura A.1b, na Figura A.1c e na Figura A.1d, respectivamente. Os mintermos destacados nos mapas de Karnaugh representam os mintermos aproximados. Na Figura A.1b, quando os mintermos $x_0 * \overline{x_1} * \overline{x_2}$ e $\overline{x_0} * x_1 * x_2$ têm a saída complementada de 0 para 1, é possível inserir os cubos x_0 e x_1. Esta inserção implica na remoção dos cubos $x_1 * \overline{x_2}$ e $x_0 * x_2$, reduzindo o número de literais de 4 para 2. Na Figura A.1c, quando os mintermos $\overline{x_0} * x_1 * \overline{x_2}$ e $x_0 * x_1 * \overline{x_2}$ têm a saída complementada de 1 para 0, é possível remover diretamente o cubo $x_1 * \overline{x_2}$, reduzindo o número de literais de 4 para 2. Na Figura A.1d, quando os mintermos $x_0 * \overline{x_1} * \overline{x_2}$ e $\overline{x_0} * x_1 * \overline{x_2}$ têm a saída complementada de 0 para 1 e de 1 para 0, respectivamente, é possível inserir o cubo x_1 e remover os cubos $x_1 * \overline{x_2}$ e $x_0 * x_2$, reduzindo o número de literais de 4 para 1.

No presente trabalho é proposta uma abordagem para a resolução do problema de 2L-ALS usando ER como restrição de erro e que utiliza a inserção e remoção de cubos de forma conjunta como técnica de aproximação. O circuito dois-níveis é representado por um cobertura $\mathcal{C}$ e o limite de ER por er. Conforme mostrado na Figura A.1, aplicar as duas técnicas em conjunto pode levar a melhores resultados do que utilizar somente a inserção de cubos, como é feito em outros trabalhos.

Inicialmente, apresentamos uma descrição geral do método de 2L-ALS proposto. Em seguida, os algoritmos para aproximar a cobertura são mostrados.

x_2 \ x_0x_1	00	01	11	10
0	0	1	1	0
1	0	0	1	1

(a)

x_2 \ x_0x_1	00	01	11	10
0	0	1	1	1
1	0	1	1	1

(b)

x_2 \ x_0x_1	00	01	11	10
0	0	0	0	0
1	0	0	1	1

(c)

x_2 \ x_0x_1	00	01	11	10
0	0	0	1	1
1	0	0	1	1

(d)

Figure A.1 – Exemplos de coberturas aproximadas comparando o uso da inserção de cubos, da remoção de cubos e de ambas em conjunto.

A.4.1 Descrição Geral

O fluxo geral é mostrado no Algoritmo 9. A cobertura é armazenada usando dois mapas da Karnaugh. O primeiro mapa liga os cubos aos mintermos que são cobertos por somente este cubo. O segundo mapa liga cada mintermo coberto pelo procedimento, contendo os cubos da solução que o cobrem. O método proposto aplica inicialmente um procedimento de inserção de cubos, gerando múltiplas coberturas parcialmente aproximadas com menos do limite de número de erros. A cobertura inicial $\mathcal{C}$ é aproximada utilizando um método de inserção com o limite de dois erros, criando duas coberturas parcialmente aproximadas (soluções parciais) com um e dois erros, respectivamente. Este comportamento é similar a primeira técnica de aceleração apresentada na Seção A.3. Para cada solução parcial, o método de remoção de cubo é aplicado, considerando o número de erros ainda disponível, que corresponde a diferença entre o erro máximo permitido e o erro já inserido na solução parcial. Esta estratégia permite escapar de mínimos locais que ocorrem quando somente a técnica de inserção é aplicada.

Para evitar um aumento substancial na complexidade de tempo e de espaço, somente a cobertura inicial e as modificações necessária para transformá-la nas soluções parciais são armazenadas. Para isso, uma solução contém os cubos que devem ser inseridos e removidos da cobertura, além da redução de literais e dos EICs inseridos. Na linha 2, o vetor de $sols$ contém todas as soluções parciais obtidas, enquanto o vetor $sols_i$ contém as soluções parciais com i erros. Desta forma, como o vetor $sols_0$ não insere nenhum

Algoritmo 9: Método de 2L-ALS Proposto

```
Entrada: Uma cobertura otimizada C e um limite de ER er
Saída: Cobertura aproximada C'
1  e ← er * 2^n ;
2  Vetor sols com e+1 vetores de Soluções;
3  sols_0 ← ∅ (solução vazia);
4  para i ← 0 até e faça
5  |  topS ← as duas melhores soluções em sols_i;
6  |  para cada solution s em topS faça
7  |  |  modifyCover(C, s);
8  |  |  (s1, s2) ← cubeInsertion(C, 2, s);
9  |  |  sols_i+1 ← sols_i+1 ∪ s1;
10 |  |  sols_i+2 ← sols_i+2 ∪ s2;
11 |  |  s3 ← cubeRemoval(C, e-i, s);
12 |  |  sMax ← max(s1, s2, s3);
13 |  |  se sMax > bestSol então bestSol ← sMax;
14 |  |  restoreCover(C, s);
15 |  fim
16 fim
17 retorna espresso(modifyCover(C,best));
```

erro na cobertura, sendo ele iniciado como vazio. Para cada vetor $sols_i$, as duas melhores soluções são selecionadas na linha 5 para serem aproximadas, similar a segunda técnica de aceleração apresentada na Seção A.3.

Para modificar a cobertura $\mathcal{C}$ considerando uma solução s é aplicada a função modifyCover na linha 7 do Algoritmo 9. Esta função insere e remove os cubos da solução da cobertura inicial, tornando-a uma cobertura aproximada. Na direção contrária, a função restoreCover desfaz estas modificações presentes. O método de inserção de cubos, na linha 8, retorna duas soluções (*s1* e *s2*) com um e dois erros que são armazenadas em $sols_{i+1}$ e $sols_{i+2}$, na linha 9 e 10. O método de remoção de cubos, na linha 11, pode inserir até $e - i$, onde e é o número máximo de erros permitidos, retornando a solução *s3*. Caso umas das três soluções geradas reduza mais literais do que a melhor solução obtida até o momento, esta solução é armazenada em $bestSol$. No final, a melhor solução encontrada é aplicada na cobertura inicial $\mathcal{C}$ e otimizada utilizando a ferramenta Espresso (BRAYTON et al., 1984).

A.4.2 Método de Inserção de Cubos

O método de inserção de cubos desenvolvido é baseado no método de busca heurística apresentado em (SU et al., 2020). Desta forma, a principal estrutura de dados para realizar a aproximação é a SICC-cube tree (SCT). A ideia geral é gerar múltiplas

SCTs com cubos que insiram até dois erros e selecionar as SCTs que levam a uma maior redução de literais.

Algoritmo 10: Função cubeInsertion

Entrada: Uma cobertura otimizada $\mathcal{C}$, um limite de número de erros e e a solução atual s

Saída: Duas soluções com erro 1 e 2

1 *trees* ← generateSCT($\mathcal{C}$, e, s.EIC);
2 (*s1*, *s2*) ← combineAndEstimate($\mathcal{C}$, *trees*);
3 *s1* ← updateSolution(*s1*, *s*);
4 *s2* ← updateSolution(*s2*, *s*);
5 retorna (*s1*, *s2*);

O Algoritmo 10 apresenta o fluxo do método de inserção de cubos. Na linha 1, as SCTs são geradas. Os cubos utilizados para a geração destas SCTs são todos os cubos obtidos através da expansão dos cubos da cobertura $\mathcal{C}$. A utilização da expansão de cubos é uma simplificação da quarta técnica de aceleração apresentada na Seção A.3. Quando um cubo expandido é usado como folha de uma SCT, é garantido que este cubo irá remover pelo menos um cubo da cobertura e irá reduzir o número de literais. Desta forma, a verificação a ser feita é relacionada ao número de erros inseridos. O número de erros inseridos por um cubo é igual ao número de mintermos cobertos por ele que não são cobertos por $\mathcal{C}$ que não tiveram a combinação de entrada considerada como EIC anteriormente.

Na linha 2, a combinação de SCTs e a estimativa da redução de literais de todas as SCTs é realizada. Inicialmente, todas as SCTs obtidas sem combinação têm a redução de literais estimada. As SCT que inserem somente um erro são combinadas dois-a-dois, considerando a quarta técnica de aceleração apresentada na seção A.3. As SCTs com um e dois erros que removem mais literais são utilizadas para criar as soluções *s1* e *s2*, que são retornadas ao final.

A.4.3 Função de Remoção de Cubos

Remover um cubo implica que os literais deste cubo são removidos da cobertura. Sendo assim, o método de remoção de cubos escolhe de forma gulosa o cubo com maior razão entre número de literais e número de erros inseridos para ser removido a cada iteração. O Algoritmo 11 mostra o fluxo deste método.

No laço da linha 3, é realizada a escolha de qual cubo será removido. Para isso, os EICs de cada cubo de $\mathcal{C}$ na linha 4 e o ganho de remover este cubo é calculado na linha

5. No contexto da remoção de cubos, o EIC de um cubo é dado pelos mintermos que somente são cobertos por este cubo em $\mathcal{C}$ que a combinação de entrada não foi adicionada como EIC anteriormente. Estes mintermos estão armazenados junto com cada cubo no primeiro mapa da estrutura de dados de $\mathcal{C}$. Caso o ganho de remover o cubo atual seja maior que o melhor ganho encontrado até o momento, o ganho, o cubo e os EICs são armazenados. Caso o cubo não tenha EICs, o ganho é maximizado. Na linha 12, o cubo com maior ganho é removido de $\mathcal{C}$. Quando um cubo é removido, os mintermos que são cobertor por somente um cubo são atualizados, impactando nas iterações seguintes. Em seguida, o vetor de cubos removidos, número de erros acumulados nesta etapa e os EICs são atualizados.

Algoritmo 11: Função cubeRemoval

```
    Entrada: Uma cobertura otimizada C, um limite de número de erros e e a
             solução atual s
    Saída: Uma solução com no máximo e erros
 1  error ← e, newEIC ← s.EIC;
 2  enquanto error > 0 faça
 3  |   para each Cube in C faça
 4  |   |   cubeEIC ← getCubeEIC(cube, C, newEIC);
 5  |   |   gain ← litCount(cube) / max(0.01, #cubeEIC);
 6  |   |   se gain > bestGain and error ≥ #cubeEIC então
 7  |   |   |   bestGain ← gain;
 8  |   |   |   bestCube ← cube;
 9  |   |   |   bestEIC ← cubeEIC;
10  |   |   fim
11  |   fim
12  |   removeCubeFromCover(bestCube, C);
13  |   removedCubes ← removedCubes ∪ bestCube;
14  |   error ← error - #bestEIC;
15  |   newEIC ← updateEICs(newEIC, bestEIC);
16  fim
17  insertCubes(C, removedCubes);
18  s3 ← updateSolution(removedCubes, newEic, s);
19  retorna s3;
```

Quando no erro máximo é atingido, o laço principal acaba. No fim, os cubos removidos são reinseridos em $\mathcal{C}$ e a solução *s3*, contendo os cubos removidos, os EICs atualizados e a redução de literais, é retornada.

A.5 Resultados Experimentais

Os algoritmos propostos foram implementados utilizando a linguagem de programação C++. Nossos experimentos foram executados utilizando o conjunto de benchmarks do IWLS'93 (MCELVAIN, 1993), em um computador com um processador quad-core i5-

2400 @ 3.10GHz e 8GB de RAM. As tabelas utilizadas nesta seção estão simplificadas. As versões originais e outras análises podem ser vistas na dissertação,

A.5.1 Comparação com o Estado-da-Arte

O método de Su (SU et al., 2020) pode ser considerado o estado-da-arte e será adotado aqui como a principal referência. Os circuitos utilizados são os mesmos utilizados por Su com uma restrição fixa de 16 erros. Como o código-fonte do método de Su não está publicamente disponível, nós comparamos os nossos resultados com os apresentados por ele no artigo (SU et al., 2020).

A Tabela A.1 mostra a comparação dos resultados obtidos pelo método de Su e do método proposto. Coluna 1 apresenta o nome dos circuitos utilizados, assim como o número de entradas (i) e saídas (o). Coluna 2 apresenta o número de literais da cobertura original, enquanto a Coluna 3 e a Coluna 4 o número de literais do circuito aproximado por Su e pelo método proposto, respectivamente. Coluna 5 apresenta a razão entre o número de literais da nossa aproximação e da aproximação de Su. Coluna 6 apresenta o número de cubos da cobertura original, enquanto a Coluna 7 e a Coluna 8 o número de cubos do circuito aproximado por Su e pelo método proposto, respectivamente. Coluna 9 apresenta a razão entre o número de cubos da nossa aproximação e da aproximação de Su. Coluna 10 e Coluna 11 apresentam os tempos de execução da cada método.

O nosso método apresentou melhores resultados para todos os circuitos utilizados, com excessão do *b12* onde nenhum dos dois métodos foi capaz de aproximar. Os circuitos *con1*, *misex1* e *b12* não foram aproximados pelo método de Su pois não existem SCTs com dois ou menos erros. Apesar disso, os circuitos *con1* e *misex1* puderam ser aproximados pelo nosso método através da etapa de remoção de cubos. Em geral, os resultados da redução no número dos cubos é similar a redução no número de literais.

O método proposto apresenta um menor tempo de execução em todos os circuitos, com um tempo de execução médio de 6,75 segundos em comparação com 643 segundos obtido por Su. Esta diferença de tempo é observada principalmente nos circuitos com mais de dez entradas, mostrando que o nosso método tem uma escalabilidade melhor.

Table A.1 – Comparação do método proposto com o método de Su (SU et al., 2020) sobre os circuitos do conjunto de benchmarks do IWLS93 considerando ER como métrica de erro.

Circuito	Literais				Cubos				Tempo (s)	
	Orig.	Su	Prop.	Prop./Su	Orig.	Su	Prop.	Prop./Su	Su	Prop.
con1 i:7;o:2	32	32	24	**0.75**	9	9	7	**0.77**	0.38	0.02
rd73 i:7;o:3	903	578	556	**0.96**	127	88	88	1.00	1.48	1.37
inc i:7;o:9	198	156	125	**0.80**	30	25	20	**0.80**	0.49	0.18
5xp1 i:7;o:10	347	235	202	**0.85**	65	49	41	**0.83**	0.72	0.48
sqrt8 i:8;o:4	188	98	83	**0.84**	38	22	21	**0.95**	0.58	0.24
rd84 i:8;o:4	2070	1578	1511	**0.95**	255	218	209	**0.95**	6.52	4.51
misex1 i:8;o:7	96	96	77	**0.80**	12	12	10	**0.83**	0.50	0.02
clip i:9;o:5	793	588	584	**0.99**	120	93	93	1.00	1.99	1.36
apex4 i:9;o:19	5419	5040	5024	**0.99**	436	421	418	**0.99**	109	31.9
sao2 i:10;o:4	496	231	165	**0.71**	58	29	22	**0.75**	2.48	1.47
ex1010 i:10;o:10	2718	2693	2636	**0.97**	284	283	278	**0.98**	14.30	2.05
alu4 i:14;o:8	5087	4904	4847	**0.98**	575	562	559	**0.99**	298	14.06
misex3 i:14;o:14	7784	7446	7242	**0.97**	690	656	635	**0.96**	693	11.99
table3 i:14;o:14	2644	2459	2347	**0.95**	175	165	159	**0.96**	513	5.47
misex3c i:14;o:14	1561	1239	1115	**0.89**	197	163	153	**0.93**	252	19.29
b12 i:15;o:9	207	207	207	1.00	43	43	43	1.00	249	1.69
t481 i:16;o:1	5233	5105	4975	**0.97**	481	473	463	**0.97**	1570	3.41
table5 i:17;o:15	2501	2410	2270	**0.94**	158	154	147	**0.95**	7868	22.07
Média	2126	1949	1888	**0.90**	208.5	192.5	186,9	**0.92**	643	6.75

A.5.2 Resultados Considerando a Frequência de Erro

Utilizar um limite de erros fixo pode ser um problema pois o ER depende do número de combinações de entrada do circuito. Por exemplo, o limite de 16 erros utilizados anteriormente representam um ER de 12.5% nos circuitos de 7 entradas mas somente 0.0001% em um circuito de 17 entradas. Como o método proposto consegue atingir um boa escalabilidade temporal, é possível utilizar um número maior de erros durante a aproximação, permitindo utilizar um limite de ER em porcentagem. A Tabela A.2 apresenta os resultados de aproximações considerando um limite de ER de 1%, 3% e 5%, utilizando os circuito utilizados anteriormente com mais de dez entradas. Coluna 1 apresenta o nome dos circuitos utilizados com o número de entradas e saídas. Coluna 2 apresenta o limite de ER, enquanto a Coluna 3 apresenta o número de erros (NE) utilizados na aproximação. Coluna 4 apresenta o número de literais do circuito original. Coluna 5 apresenta o número de literais do circuito aproximado pelo método proposto e a porcentagem de redução entre parênteses. Coluna 6 apresenta o número de cubos do circuito original. Coluna 7 apresenta o número de cubos do circuito aproximado pelo método proposto e a porcentagem de redução entre parênteses. Coluna 8 apresenta o tempo de

execução.

Table A.2 – Resultados do método proposto sobre os circuitos do conjunto de benchmarks do IWLS93 considerando ER como métrica de erro.

Circuitos	ER	NE	Literais		Cubos		Tempo (s)
			Orig.	Aproximado	Orig.	Aproximado	
sao2	1%	10	496	273 (0.55)	58	33 (0.56)	0.83
i: 10	3%	30		75 (0.15)		12 (0.20)	3.14
o: 4	5%	51		31 (0.06)		7 (0.12)	4.74
ex1010	1%	10	2718	2659 (0.97)	284	279 (0.98)	1.27
i: 10	3%	30		2588 (0.95)		273 (0.96)	3.92
o: 10	5%	51		2510 (0.92)		267 (0.94)	6.93
alu4	1%	163	5087	3732 (0.73)	575	461 (0.80)	148.22
i: 14	3%	491		2693 (0.52)		356 (0.62)	286.48
o: 8	5%	819		2139 (0.42)		297 (0.51)	429.81
misex3	1%	163	7784	6253 (0.80)	690	554 (0.80)	154.03
i: 14	3%	491		4796 (0.61)		436 (0.63)	217.58
o: 14	5%	819		3749 (0.48)		350 (0.50)	253.85
table3	1%	163	2644	1271 (0.48)	175	93 (0.53)	32.59
i: 14	3%	491		536 (0.20)		42 (0.24)	36.56
o: 14	5%	819		189 (0.07)		17 (0.10)	44.44
misex3c	1%	163	1561	499 (0.32)	197	89 (0.45)	133.55
i: 14	3%	491		469 (0.30)		85 (0.43)	229.03
o: 14	5%	819		447 (0.28)		82 (0.41)	342.40
b12	1%	372	207	193 (0.93)	43	40 (0.93)	2.34
i: 15	3%	983		167 (0.80)		35 (0.81)	7.52
o: 9	5%	1638		145 (0.70)		28 (0.65)	15.71
t481	1%	655	5233	1992 (0.38)	481	212 (0.44)	4.67
i: 16	3%	1966		942 (0.18)		120 (0.24)	6.00
o: 1	5%	3276		578 (0.11)		84 (0.17)	8.85
table5	1%	1310	2501	720 (0.28)	158	55 (0.34)	152.81
i: 17	3%	3932		278 (0.11)		24 (0.15)	302.28
o: 15	5%	6553		152 (0.06)		14 (0.08)	375.89

O método proposto alcança uma redução de literais média de 38% com ER de 1%, 56% com ER de 3% e 64% com ER de 5%. Nos circuitos *sao2*, *table3*, *t481* e *table5*, foi possível obter perto de 90% de redução de literais com 5% de ER e de 93.9% no circuito *table5* com o mesmo ER. Além disso, o circuito *b12* que não pôde ser aproximado com 16 erros, foi aproximado neste experimento, devido ao número maior de erros permitidos, alcançando até 26.1% e redução de literais. Mesmo nos circuitos com um número maior de erros devido ao número de entrada, o método de manteve escalável, executando em menos de 5 minutos.

A.6 Conclusão

Com o uso crescente de aplicações resilientes a erro, é de interesse o projeto de circuitos integrados utilize esta característica para otimizar área, desempenho e consumo energético dos circuitos. Desta forma, é importante focar no desenvolvimento de ferramentas para síntese de circuitos aproximados otimizados e que sejam escaláveis.

Neste trabalho, foi proposto um método escalável para síntese de circuitos aproximados dois-níveis considerando a métrica ER que utiliza as técnicas de inserção e remoção de cubos. Os resultados experimentais mostraram que o trabalho proposto supera o método estado-da-arte em qualidade de resultados e em escalabilidade. O código-fonte desenvolvido, os circuito originais utilizados e os circuitos aproximados gerados estão disponíveis no GitHub [1].

Pretende-se estender este trabalho em duas direções futuras:

- Modificar o método utilizado para gerar circuitos aproximados considerando diferentes métricas que adotem a magnitude do erro. O uso dessas métricas é interessante para a aproximação de circuitos aritméticos.
- Utilizar o trabalho proposto como um "aproximador local" de um método ALS multinível. Ou seja, o uso de um método de ALS dois-níveis para aproximar subcircuitos de um circuito multinível pode ser relevante.

[1] <https://github.com/GabrielAmmes/2LALS-IR>