FRAMEWORKS AND METHODOLOGIES FOR SEARCH-BASED APPROXIMATE LOGIC SYNTHESIS

DISSERTATION

A thesis submitted to the
FACULTY FOR COMPUTER SCIENCE, ELECTRICAL ENGINEERING AND MATHEMATICS
of
PADERBORN UNIVERSITY
in partial fulfillment of the requirements
for the degree of *Dr. rer. nat.*

by

LINUS MATTHIAS WITSCHEN

Paderborn, Germany
Date of submission: June 2022

SUPERVISOR:
Prof. Dr. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand
Prof. Dr. Laura Pozzi

ORAL EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand
Prof. Dr. Laura Pozzi
Prof. Dr. Christian Plessl
Dr. Michael Laß

DATE OF SUBMISSION:
June 2022

Dedicated to my parents.

And they that are wise shall shine
as the brightness of the firmament;
and they that turn many to righteousness
as the stars for ever and ever.

— Daniel 12:3

ACKNOWLEDGMENTS

Many people supported me in realizing my dissertation and enabled me to achieve my goals. First and foremost, I would like to express my deepest gratitude to Prof. Dr. Marco Platzner for his support and guidance during my entire research, as well as for teaching me that good research is more than just academic skills. Furthermore, I want to thank him, Prof. Dr. Sybille Hellebrand, and Prof. Dr. Laura Pozzi for the time and effort they spent reviewing this dissertation and all committee members for evaluating my work.

I am indebted also to all my colleagues and co-workers who kept me sane through well-timed coffee breaks and kept me motivated by daydreaming of changing the future through research. Here, I want to note my colleagues from approximate computing: Hassan Ghasemzadeh Mohammadi and Muhammad Awais, who helped me advance my research, and especially Tobias Wiersema for guiding me through my Master's thesis and the many fruitful discussions. Moreover, I would like to recognize my student research assistants, project group members, and the students writing their thesis with me for contributing to my research projects.

Special thanks to Angela Gutierrez for slipping into the role of a rubber duck and helping me solve many of my research challenges by listening. You have been a pillar on my journey, and your affection made me forget about stressful times.

Finally, I am incredibly grateful to my family for their encouragement and support. Especially, I would like to thank my mother for her endless support and never ending care. In my profound appreciation, I esteem my late father's guidance throughout my (academic) life. He shaped me as a person and has always been a role model; following his advice, I did as well as I could...

ABSTRACT

Approximate computing has emerged as one way to meet the challenge of improving a computing system's performance by trading off an application's quality against a target metric. This dissertation focuses on approximate computing at hardware level, where it is referred to as approximate logic synthesis (ALS) and has the goal of generating approximate circuits; specifically, this dissertation makes five contributions and comprehensively considers automated search-based ALS processes that are modeled with four main steps: *search*, *approximate*, *verify*, and *estimate*.

Firstly, this dissertation contributes the CIRCA framework that implements a general and fully configurable ALS process to provide an environment for comparing different ALS methods. Secondly, we propose the jump search methodology that minimizes syntheses and verifications by exploiting domain knowledge to rapidly generate approximate circuits. Thirdly, the technique MUSCAT contributes to the approximation step and utilizes formal verification techniques to construct approximate circuits that are valid-by-construction regarding their quality. The fourth contribution considers the verification step and is the concept of proof-carrying approximate circuits, which brings together the fields of approximate computing and proof-carrying hardware. Finally, this dissertation proposes a formal verification-based methodology that characterizes the search space of approximate circuits prior to the ALS process.

ZUSAMMENFASSUNG

Approximate computing hat sich als ein Weg herauskristallisiert, die Verarbeitungsleistung von Rechensystemen weiter zu steigern, indem die Qualität einer Anwendung gegen eine Zielmetrik eingetauscht wird. Diese Dissertation betrachtet approximate computing auf der Hardwareebene, auf der es approximierte Schaltungen generiert und als approximate logic synthesis (ALS) bezeichnet wird. Konkret leistet die Arbeit fünf Beiträge und betrachtet automatisierte, suchbasierte ALS Prozesse, die in vier Schritten modelliert werden: *suchen*, *approximieren*, *verifizieren* und *abschätzen*.

Zunächst stellt die Dissertation das Framework CIRCA vor, das den ALS Prozess allgemeingültig und konfigurierbar implementiert und so eine Umgebung für den Vergleich von ALS Methoden bereitstellt. Anschließend wird das Suchverfahren jump search diskutiert, das approximierte Schaltungen schnell generiert, indem es Synthesen und Verifikationen mittels Domänenwissen minimiert. Des Weiteren wird die Approximationstechnik MUSCAT vorgestellt, die auf formaler Verifikation basiert und Schaltungen hinsichtlich ihrer Qualität korrekt konstruiert. Ferner betrachtet die Arbeit den Verifikationsschritt und stellt das Konzept von proof-carrying approximate circuits vor, das die Felder approximate computing und proof-carrying hardware vereint. Abschließend behandelt die Arbeit ein auf formaler Verifikation basierendes Verfahren, das vorab den Suchraum von approximierten Schaltungen für den ALS Prozess charakterisiert.

AUTHOR'S PUBLICATIONS

[1] Alexander Boschmann, Andreas Agne, Georg Thombansen, Linus Witschen, Florian Kraus, and Marco Platzner. "Zynq-based acceleration of robust high density myoelectric signal processing." In: *Journal of Parallel and Distributed Computing* 123 (2019), pp. 77–89. DOI: 10.1016/j.jpdc.2018.07.004.

[2] Alexander Boschmann, Andreas Agne, Linus Witschen, Georg Thombansen, Florian Kraus, and Marco Platzner. "FPGA-based acceleration of high density myoelectric signal processing." In: *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. [Best New Applied Domain Paper Award]. IEEE, 2016. DOI: 10.1109/reconfig.2015.7393312.

[3] Alexander Boschmann, Georg Thombansen, Linus Witschen, Alex Wiens, and Marco Platzner. "A Zynq-based dynamically reconfigurable high density myoelectric prosthesis controller." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017. DOI: 10.23919/DATE.2017.7927137.

[4] Linus Witschen. "A Framework for the Synthesis of Approximate Circuits." Master's Thesis. Paderborn, Germany: Paderborn University, Aug. 2017.

[5] Linus Witschen. *CIRCA – A Modular and Extensible Framework for Approximate Circuit Generation*. [Online]. 2018. URL: https://go.uni-paderborn.de/circa.

[6] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." In: *Microelectronics Reliability* 99 (2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

[7] Linus Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. "Jump Search: A Fast Technique for the Synthesis of Approximate Circuits." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2019.

[8] Linus Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. "Jump Search: A Fast Technique for the Synthesis of Approximate Circuits." In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2019. DOI: 10.1145/3299874.3317998.

[9] Linus Witschen, Tobias Wiersema, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2018.

[10] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Making the Case for Proof-carrying Approximate Circuits." Workshop on Approximate Computing (WAPCO). Workshop without proceedings. 2018. URL: https://api.semanticscholar.org/CorpusID:52228901.

[11] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Proof-carrying Approximate Circuits." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28 (9 2020), pp. 2084–2088. DOI: 10.1109/TVLSI.2020.3008061.

[12] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Search Space Characterization for AxC Synthesis." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2020.

[13] Linus Witschen, Tobias Wiersema, Masood Raeisi Nafchi, Arne Bockhorn, and Marco Platzner. "Timing Optimization for Virtual FPGA Configurations." In: *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*. Springer Lecture Notes in Computer Science, 2021.

[14] Linus Witschen, Tobias Wiersema, Lucas David Reuter, and Marco Platzner. "MUSCAT: MUS-based Circuit Approximation Technique." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 172–177. DOI: 10.23919/DATE54114.2022.9774604.

[15] Linus Witschen, Tobias Wiersema, Lucas David Reuter, and Marco Platzner. "Search Space Characterization for Approximate Logic Synthesis." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2022. DOI: 10.1145/3489517.3530463.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

LIST OF TABLES

1

INTRODUCTION

With the breakdown of Dennard scaling at the beginning of the century and the continuous slow-down – and eventually the end – of Moore's law, chip designers are challenged to find new ways to increase processing performance under strict power and area constraints. The design paradigm approximate computing (AC) is one way of meeting the challenge.

Approximate computing exploits the fact that many domains show an inherent resilience against inaccuracies and even errors to trade off computational accuracy (or quality) against target metrics, e.g., hardware area, delay, or power consumption. Mittal [58] describes the trade-off as the gap between the level of *required* accuracy by the user and the *provided* accuracy by the computing system. Research groups from academia [24] and industry [17, 33, 57, 63] have shown that this gap can be found in numerous domains, including signal processing, audio, image and video processing, machine learning, and data analytics. For example, Chippa et al. [24] reported that representative applications from the domain of recognition, data mining, and search spend on average 83% of the runtime on resilient computations; thus, demonstrating the potential that approximate computing offers. An application's resilience against inaccuracies and errors can be attributed to several factors [24, 58, 82, 112]:

- *Redundant input data*: Redundancy in the input data makes the application more robust against imprecisions.
- *Error attenuation*: Computational patterns, such as iterative refinement or statistical aggregation, in the application's algorithm increase the resilience; thus, facilitating the processing of noisy, imprecise, or incomplete input data, e.g., data from physical sensors such as cameras or microphones.
- *Lack of golden result*: Rather than a golden result (or correct answer), a range of acceptable results exists, e.g., recommender systems or web searches.
- *Perceptual limitations*: Visual and audio perception of humans, for example, is limited and slight differences of individual pixels cannot be recognized.

Figure 1.1 shows a practical example of approximate computing, demonstrating the resilience of an application and the potential of approximate

computing. Gupta et al. [38] replaced the precise adder cells with approximate counterparts within a hardware module for JPEG compression and, in this way, achieved power savings of around 53%. The left-hand image shows the original, precise compression outcome; the right-hand image shows an outcome produced by the approximate hardware. Compared to the precise outcome, the quality of the approximate outcome is reduced, as indicated by the lower peak signal-to-noise ratio (PSNR). However, the differences are merely visible due to perceptual limitations, and the approximate outcome is almost indistinguishable from the original, which justifies the acceptance of the suboptimal quality in return for the significantly reduced power consumption.

(a) Reference (b) Approximate outcome

Figure 1.1: Demonstration of approximated JPEG compression. Taken from [38].

In fact, suboptimal quality of results and limited accuracy have always been present in computer science and engineering due to limited precision of data types, the use of (meta-)heuristics, or the exact or optimal result being out of reach. Approximate computing, however, goes further and exploits the resilience by applying approximations through the judicious introduction of errors to leverage the trade-off of quality and target metrics. Here, the approximations are complementary [25, 113, 119] and can be applied to all system components (processing, storage, and communication) at all levels of the computing stack (from applications down to semiconductor technology) [58, 82, 112]. Figure 1.2 shows the computing stack assumed in this dissertation that divides into a software and a hardware part.

The *application/algorithm* level represents the highest level of abstraction, and approximations target the application's algorithm via source-code transformations. Precision scaling (or quantization), for example, states a well-known, general approximation technique that reduces a variable's bit width to simplify computations and reduce the memory footprint. Other techniques target specific applications such as neural networks to improve the network's energy efficiency by approximating resilient neurons using, for instance, precision scaling or an approximated activation function [93].

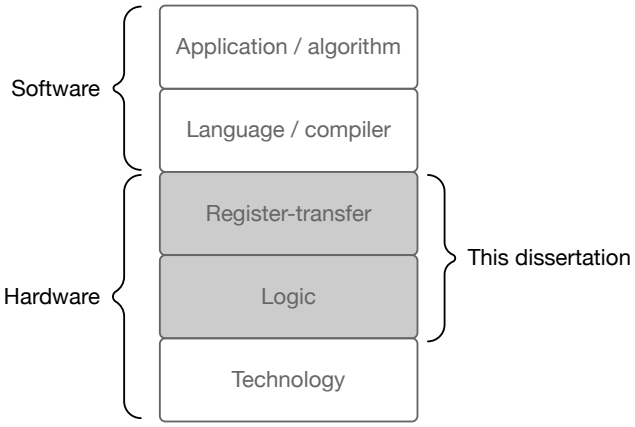


Figure 1.2: Computing stack with focus of this dissertation highlighted in gray.

On the *language/compiler* level, programming or hardware description languages have been extended to allow developers to specify parts of the code that are amenable to approximations. *EnerJ* [72], for example, extends Java by approximate data types that enable the developer to declare which data can be processed by approximate processing units or stored in approximate storage. Compared to EnerJ, *Axilog* [115] locates at a lower level and extends the hardware description language Verilog by a set of language annotations to denote parts of the design for which relaxed quality constraints apply. For languages, the designer manually specifies the parts that may be subjected to approximation, what approximations are eventually applied, however, is decided by the compiler [18] or synthesis flow [69, 101] and also depends on the underlying approximate hardware which can comprise quality programmable processors [92], approximate hardware accelerators [34], and approximate storage [73]. General approximation techniques used by compilers are loop perforation [83], which reduces the loop count by skipping iterations, and code perforation [2], which skips computations in the algorithm. Both techniques reduce the computational workload, and thus, the application's computation time and the overall energy consumption.

On the hardware level, approximate computing is denoted as approximate logic synthesis (ALS) [75], which describes the process of generating approximate circuits (AxCs) for approximate hardware. The *register-transfer level* represents the most abstract hardware level in our computing stack and, at this level, precise components are replaced by approximate counterparts – ranging from simple components to complex hardware accelerators [34] or even complete processors [92]. Components subjected to approximations are denoted as *candidates*, and ALS at register-transfer level (RTL) seeks to find a combination of approximate candidates that optimizes the target metrics. Hence, ALS at RTL becomes a combinatorial problem where exhaustive enumeration is infeasible due to the large number of candidates that real-world hardware designs provide. Thus, automated ALS generally employs search or optimization algorithms to explore the design space [65, 69, 102] and approach the problem systematically. For the approximate candidates,

ALS can resort to approximate component libraries [44, 45, 62, 70, 81, 88], approximation techniques that operate on the candidate's implementation at lower levels of abstraction [22, 41, 50, 91, 94, 106], or precision scaling to truncate least significant bits [69, 86, 101].

On the *logic* level, ALS modifies a circuit's Boolean representation or transforms a circuit's gate level netlist. Boolean techniques operate on abstract data structures, e.g., truth tables, and-inverter-graphs, or binary decision diagrams, and apply transformations to simplify the logic through Boolean matrix factorization [41], tying signals to constant values [22, 35], or extending the circuit's set of don't cares by approximate don't cares [91]. Approaches transforming the circuit's gate level netlist operate on a lower level of abstraction than Boolean approaches since the approximations target the description of the circuit's electrical components and their connections. Pruning gates [77], substituting signals to enforce simplification [94], or random gate replacements [50] represent common netlist transformation techniques.

The *technology* level represents the lowest level of abstraction where transistors are – even manually – removed from a logic gate [38, 54] or timing-induced approximations are introduced [117, 118]. Timing-induced approximations involve over scaling a circuit's supply voltage and/or the operating frequency, meaning that the circuit operates at a supply voltage below its nominal value or an operating frequency above its maximum, respectively. The effects of the timing-induced approximations are immediate since the quadratic relationship between voltage and power is exploited, or the circuit produces its output with less delay. However, timing-induced approximations require an expensive and complex timing-aware analysis of the circuit to detect and evaluate the failing timing paths; otherwise, the errors become uncontrollably large [75, 86, 112].

This dissertation focuses on approximate computing at hardware level, i.e., approximate logic synthesis; specifically, the focus is on automated search-based ALS processes on the register-transfer level and logic level, as highlighted in Figure 1.2. At these levels, the approximations deliberately modify the circuit's original function and range from fine-grained to coarse-grained, i.e., from gates at logic level to more abstract components at RTL, while the provided abstraction allows for an efficient analysis of the circuit's quality and target metrics.

Modeling approximate logic synthesis as a search or an optimization problem is common to perform the approximation systematically, and four main steps can be defined to describe a general search-based ALS process: *search*, *approximate*, *verify*, and *estimate*. This dissertation makes methodological contributions to three of the four steps (search, approximate, and verify). Furthermore, the dissertation contributes the flexible ALS framework CIRCA and a novel pre-processing technique for search space characterization. In summary, the main contributions of this dissertation are:

- *CIRCA* (Chapter 3, [101, 103]): We design and implement the framework CIRCA for search-based approximate logic synthesis. In a detailed analysis, we classify existing ALS frameworks on the basis of a set of orthogonal categories. Our analysis shows that existing frameworks suffer from an inflexible design that severely hampers the development and evaluation of new techniques for AxC generation, as well as the comparison to existing ones. Using the analysis, we identify key requirements that an ALS framework should fulfill, yet the existing frameworks lack. With CIRCA, we present the first ALS framework that meets all requirements, and thus, presents a flexible framework to the approximate computing community to facilitate comparability among ALS methods.
- *Jump search (Chapter 4, [102, 108])*: We propose jump search, a novel technique for rapidly synthesizing approximate circuits and our contribution to the search step. ALS usually explores a large number of AxCs, but the required syntheses and verifications are costly; thus, forming a bottleneck for ALS. Jump search seeks to minimize the invoked syntheses and verifications by first selecting a set of AxCs-of-interest from the search space and then only evaluating a subset of the selected AxCs via synthesis and verification. The initial selection bases on a heuristic function that is free of synthesis and verification. Instead, the heuristic incorporates pre-computed *impact factors* that encode information on a candidate's impact on the target metric and the overall circuit error. In our experimental results, we show speed-ups of up to 468× while achieving improvements in the target metrics comparable to commonly-used search algorithms.
- *MUSCAT* (Chapter 5, [106]): We present MUSCAT, a technique to approximate netlists and our novel contribution to the approximation step. MUSCAT substitutes connections with constant values by activating *cutpoints*. In this way, MUSCAT enables synthesis tools to simplify the logic by pruning dangling gates and propagating constants. Utilizing the concept of minimal unsatisfiable subsets from the field of formal verification, MUSCAT determines a maximal number of activated cutpoints and generates AxCs that are *valid-by-construction* regarding their quality constraints. We show that MUSCAT achieves up to 80% higher savings in the target metric hardware area than a state-of-the-art technique.
- *Proof-carrying approximate circuits* (Chapter 6, [104, 105]): The concept of proof-carrying approximate circuits brings together the fields of approximate computing and proof-carrying hardware, and is a novel contribution to the verification step. We utilize the concept of proof-carrying hardware [30] to annotate approximate circuits with *proof certificates*. The proof certificates allow the verification of the AxCs' quality without referring to a full formal verification of the quality constraints. A full verification generally shows long runtimes and is required to overcome trust issues between a producer and a consumer of an AxC. Using an AxC's proof certificate, however, allows the con-

sumer to verify the quality of the AxC quickly without having to trust the AxC producer; in fact, the burden of the full formal verification and the generation of the proof certificate is on the producer's side, which has the computational resources to carry out the task efficiently and economically – in contrast to the consumer. In our experimental results, we reduce the consumer's runtime for verifying an AxC by up to 99.83% over the producer's runtime.

- *Search space characterization* (Chapter 7, [107, 109]): We present a formal verification-based methodology that performs *prior to* the ALS process and characterizes regions of the search space that only contains valid AxCs, i.e., AxCs that meet the quality constraints. We propose a novel *approximation miter* that makes our methodology independent of subsequently applied approximations and guarantees the validity of the as-valid characterized regions through the formal verification approach. Thus, the subsequent ALS process can employ any approximation technique and safely omit verifications for AxCs falling into the valid regions. In our experimental results, our approach achieves speed-ups of up to 3.7× over a standard ALS that requires verifying all explored AxCs.

The remainder of this dissertation structures as follows:

Chapter 2 introduces the underlying concepts of this dissertation and discusses the state-of-the-art in the domain of approximate logic synthesis. In more detail, Section 2.1 describes the concepts of a general search-based ALS and discusses the state-of-the-art ALS methods on the register-transfer level and logic level. Section 2.2 elaborates on methods for the quality assurance of AxCs.

Chapter 3 presents CIRCA, our flexible ALS framework. Section 3.2 provides an analysis and a classification of existing ALS frameworks, Section 3.3 identifies key requirements for a flexible ALS framework, and Section 3.4 discusses CIRCA's concept and architecture in detail. Section 3.5 evaluates CIRCA's experimental results.

Chapter 4 proposes our synthesis methodology jump search. Section 4.2 motivates jump search's concept, Section 4.3 presents the methodology, and Sections 4.5 and 4.6 discuss the different impact factors used in the heuristics presented in Section 4.7. Sections 4.8 and 4.9 detail jump search's implementation and discuss the experimental results.

Chapter 5 presents our approximation technique MUSCAT. Section 5.2 discusses MUSCAT's methodology, Section 5.3 presents comprehensive experimental results, and a case study in Section 5.4 probes MUSCAT's potential at higher levels of abstraction, i.e., at register-transfer level.

Chapter 6 discusses the concept of proof-carrying approximate circuits. Section 6.2 elaborates on proof-carrying hardware, Section 6.3 discusses the

approach, Section 6.4 details the verification-based ALS process and the proof certificates, and Section 6.5 presents the experimental results.

Chapter 7 presents our methodology for characterizing valid regions in the search space prior to ALS. Section 7.2 discusses the related work and describes our novel approach. Then, Sections 7.3.1 and 7.3.2 detail on the construction of our novel approximation miter and Section 7.3.3 presents our methodology, before Section 7.4 evaluates the experimental results.

Chapter 8 concludes the dissertation, and Chapter 9 discusses future work.

2

BACKGROUND

In the previous chapter, we have provided an introduction to the field of approximate computing and have set the focus for this thesis on approximate logic synthesis. In this chapter, we describe a general approximate logic synthesis process and discuss the state-of-the-art in this field. Then, we elaborate on methods for assuring the quality of the approximate outcome generated during the approximate logic synthesis process.

2.1 APPROXIMATE LOGIC SYNTHESIS

In this section, we first provide an overview of approximate logic synthesis in general and elaborate on the focus of this dissertation, i.e., automated search-based approximate logic synthesis. Then, we discuss the related work and the state-of-the-art, following in parts our published analyses [101, 106].

2.1.1 *Overview*

Approximate computing (AC) on the hardware level is also referred to as approximate logic synthesis (ALS) [75]. ALS starts from an exact circuit design and applies approximations to optimize one or multiple target metrics, e.g., hardware area, delay, or power consumption. The approximations are applied to subcircuits or components amenable to approximations, so-called *candidates*. Candidates can be complex processing units or cuts in a gate level netlist and can be identified manually [115] or automatically [37, 65]. In order to ensure that an approximate circuit (AxC) provides sufficient quality to an application, the user imposes quality constraints upon the circuit's primary outputs (POs). Consequently, satisfying the user-defined quality constraints is a primary concern for an AxC. If an AxC adheres to the constraints, the AxC is considered *valid*; otherwise, the AxC is considered *invalid*. An input design provides many possible approximation opportunities, and existing work on ALS seeks to generate optimal AxCs either via analytical methods [42, 49, 79, 80] or search-based methods [20, 50, 65, 69, 76, 94].

Analytical methods are developed for a specific set of approximation techniques and error metrics, making the approach less flexible but generally runtime-efficient. Vašíček [90] additionally observed that analytical models at gate level might be fast, but increasing model granularity increases the complexity of their derivation due to non-trivial conditional probabilities.

Thus, constructing accurate yet simple mathematical models is currently impossible. Search-based methods, on the other hand, are more flexible and support a general set of approximation techniques and error metrics. However, search-based methods usually endure longer runtimes due to the increased generality.

This dissertation considers automated search-based ALS on the register-transfer level (RTL) and logic (or gate) level, where the original circuit design is iteratively approximated. In general, search-based ALS executes the following four main steps:

1. *Search* to explore the search space of approximate circuits.
2. *Approximate* to generate approximate circuits.
3. *Verify* to ensure that the approximate circuit satisfies the user-defined quality constraints and is valid.
4. *Estimate* (or *evaluate*) to determine the approximate circuit's target metrics.

In search-based ALS, a search or optimization algorithm explores the search space of AxCs and acts as central controller of the ALS process. In each iteration, the search explores new AxCs that are generated in an approximation step. The generated AxCs are then evaluated for the target metrics; a quality assurance step verifies the AxC's validity.

Scarabottolo et al. [75] classify the related work on ALS into three categories: 1) approximate high-level synthesis (AHLS), 2) Boolean rewriting, and 3) netlist transformation. The categories mainly differ in the level of abstraction and can complement each other [82, 86, 113, 119].

AHLS operates on the highest level of abstraction in ALS, the behavioral RTL, where designs are described, for example, in C/C++ or behavioral RTL Verilog. Both Boolean rewriting and netlist transformation locate on the logic level. Boolean rewriting approximates a Boolean representation of a design, e.g., truth tables or and-inverter-graphs (AIGs), and thus, operates on a more abstract level than netlist transformation that approximates a circuit's gate level netlist. Sections 2.1.2 to 2.1.4 provide an overview of prominent techniques and methods from the three categories. For more details, we refer the interested reader either to the corresponding publication of the method or to the survey of Scarabottolo et al. [75].

The three categories comprise methods that apply functional approximations, i.e., the implemented functionality of the circuit is simplified by introducing errors to the computations judiciously. In contrast to functional approximations stand timing-induced approximations applied on the technology level, which are out of the scope of this dissertation yet briefly explained for the sake of completeness.

Timing-induced approximations [68, 117, 118] are caused by voltage over-scaling or over-clocking and can be applied to any digital circuit [89]. With over-clocking, the circuit operates above its maximum operating frequency to lower the circuit's delay. Voltage over-scaling operates a circuit at a supply

voltage below its nominal value while maintaining the operating frequency. By exploiting the quadratic relationship between the supply voltage and the dynamic power dissipation, voltage over-scaling thus achieves power savings. Path delays, however, increase with decreasing supply voltage which, in turn, leads to timing errors. The timing-induced errors are extremely difficult to predict since the actual timing errors depend on the chip's design and layout. Thus, expensive timing-aware simulations on the final chip design must be performed for a thorough and reliable analysis, leading to a very time-consuming ALS process. In fact, without a systematic analysis, the errors can become uncontrollably large [75, 86, 112]. Furthermore, timing-induced approximations often achieve relatively small energy efficiency gains since circuits often contain many near-critical paths that the timing errors affect [60, 112]. Thus, most proposed approximation methods resort to functional approximations [112].

2.1.2 *Approximate High-level Synthesis*

Approximate HLS integrates approximate operators as building blocks, i.e., substitutes original candidates with approximate variants. To substitute the candidates, AHLS can utilize approximate candidates from a pre-generated approximate component library [45, 55, 62, 88] or employ dedicated approximation techniques from the logic level [22, 41, 56, 77, 91, 121, 122] to generate the approximate candidates during the ALS process. By employing methods from the lower levels, AHLS becomes, in a sense, an over-arching ALS process, which breaks ALS down into sub-problems that are solved on the lower levels, e.g., through Boolean rewriting methods or netlist transformations, and stitched back together in AHLS.

Nepal et al. [64, 65] proposed the ABACUS methodology that transforms a circuit into an abstract synthesis tree (AST). In an iterative approach, transformations on the AST are applied to create approximate circuits. The accuracy of the AxCs is evaluated by testing, area and power characteristics via ASIC synthesis using a standard cell library. The resulting three metrics are then combined into a fitness function, and the AxC with the best fitness is greedily selected as the starting circuit for the next iteration. This heuristic process runs for a user-defined number of iterations. Eventually, a Pareto front of designs is given, trading off accuracy for power.

The ASLAN framework [69] by Ranjan et al. is, to the best of our knowledge, the only framework able to approximate sequential circuits while guaranteeing error bounds. In a first step, ASLAN extracts combinational subcircuits amenable to approximation, i.e., the candidates. Then, a search space is generated by creating approximated versions of the candidates that vary in their local error constraints and estimated energy consumption. The applied approximation techniques are precision scaling and SALSA [91], although the authors also mention the applicability of other techniques. Finally, ASLAN employs hill climbing to find a locally optimal combination of approximated candidates. In each iteration, candidate versions with larger error

bounds are considered, and the combination resulting in the most significant energy savings is selected if the circuit adheres to the global error bound. Otherwise, the next-best combination of candidates is selected. Quality assurance relies on a so-called *sequential quality constraint circuit* that raises a flag if the error bound is violated. Since ASLAN deals with sequential circuits, time frame expansion is used to unroll both the original and the approximate circuit until they finish their computations. The resulting Boolean expression is then formally verified with a satisfiability (SAT) solver.

Barbareschi et al. [10] proposed the IDEA framework that employs a branch-and-bound algorithm. In each iteration, IDEA utilizes a depth-first search strategy and examines all approximation possibilities for a target candidate until no more approximations are possible, i.e., no further approximations can be applied to the candidate or the resulting AxC violates the user-defined quality constraints. Then, IDEA backtracks and explores the remaining branches until the search budget is exhausted. The branch-and-bound algorithm solely considers the quality of an AxC and IDEA only determines circuit parameters, such as area and power consumption, once all AxCs are found that meet the quality constraints. In this way, IDEA spends the largest amount of the search budget on identifying the best possible approximation of the early selected candidates in the design.

2.1.3 *Boolean Rewriting*

Boolean rewriting approximates the logical or Boolean representations of a circuit, such as truth tables or AIGs. The methodology BLASYS [41] relies on Boolean matrix factorization and factorizes a circuit's truth table into two smaller truth tables, which results in a smaller AxC after synthesis. Considering complete truth tables, however, limits BLASYS to small combinational circuits. For larger circuits, the authors propose to firstly factorize subcircuits for every factorization degree. Then, by integrating the approximated subcircuits into the overall design, BLASYS explores the search space until the user-defined error threshold is reached.

ALSRAC [56] by Meng et al. relies on approximate care sets, determined via logic simulation and expressed for internal nodes. Using the approximate care patterns, ALSRAC generates local approximate changes (or approximate resubstitutions), which are applied iteratively until the error threshold is reached.

Venkataramani et al. presented SALSA [91] which forms a so-called *quality constraint circuit* by providing the original and the approximate circuit, which initially is identical to the original circuit, with the same input and feeding the outputs of the circuits into a quality function that checks whether the given error bound holds. Forcing the error bound to hold, SALSA works backwards and applies standard don't care logic optimization techniques to reduce the area of the approximate circuit. Thus, the technique creates approximate combinational circuits that adhere to the error bound by construction.

Similar to SALSA, the approach of Chandrasekharan et al. [22] employs a setup with a quality constraint circuit but formally verifies the error constraint by combinational equivalence checking using a SAT solver. The approach represents a circuit's logic function as AIG and employs approximation-aware AIG rewriting as approximation technique, i.e., setting nodes to constant zero. Among all possible cuts on the critical paths of the circuit, the one with the smallest cut size is selected for rewriting. This heuristic is greedily iterated until there is no more possibility for rewriting without violating the error bound or the maximum number of iterations is reached.

Soeken et al. [85] represent the circuit description as a binary decision diagram (BDD) and apply approximations to reduce the size of the BDD. Fröhlich et al. [35] also utilize a BDD representation to determine an optimal BDD, i.e., a BDD with a minimum number of nodes, satisfying the quality constraints. Using BDDs allows the authors to make precise statements about the resulting error. However, the major weakness of these approaches is that BDDs are only applicable to small combinational circuits, i.e., BDDs do not scale well for large Boolean functions.

2.1.4 *Netlist Transformation*

Netlist transformation techniques operate on a circuit's gate level netlist, and a commonly-used approach is gate level pruning (GLP) presented by Schlachter et al. [77, 78]. GLP disconnects a wire from the node driving it and inserts a constant as driver instead. As a result, a synthesis tool can optimize the logic by 1) removing (or pruning) dangling nodes and 2) simplifying the subsequent logic through constant propagation. To determine which node to prune, the authors rank the nodes by the product of their switching activity and their significance, a measure for the node's impact on the circuit's error. An iterative algorithm simulates the hardware design to compute each node's significance-activity product and evaluates whether the design meets the quality constraints. The algorithm then prunes the node with the lowest significance-activity product from the design in the next iteration. However, Scarabottolo et al. [74, 76] showed that a node's significance is often too conservative, leading to suboptimal AxCs.

Thus, Scarabottolo et al. [76] extended GLP to circuit carving. Circuit carving seeks to carve out the largest subcircuit in the design so that the resulting AxC still meets the quality constraints. The authors propose to use a node's significance as an error estimate and explore a binary search tree to determine whether a node is included in the subcircuit. However, as an AxC's actual error cannot be determined via the significances, a subsequent quality check is performed. An exploration of the complete binary search tree is considered intractable; hence, the authors defined pruning criteria for the tree. One criterion considers a node's estimated significance (or estimated error). To estimate the nodes' significance, the authors suggest simulating the complete circuit exhaustively, which is accurate but only applicable to small circuits, or employing an error estimation model, which has shown to

be often too conservative. In any case, the error estimation's accuracy dictates the approximate outcome.

In an attempt to increase the error estimation accuracy and improve the quality of results in GLP, Scarabottolo et al. [74] proposed Partition & Propagate (P&P). P&P partitions the design into cuts which are simulated exhaustively to determine each node's significance. Due to the exhaustive simulation, the accuracy of the nodes' significance becomes more accurate. The authors argue that the number of inputs to the cuts is small, and thus, simulation can be performed efficiently. Nevertheless, the approach can only specify an estimate of an error bound, which, as their experimental results show, may still be too conservative by several orders of magnitudes. Consequently, a subsequent quality verification is required.

Another netlist transformation approach is circuit design by evolutionary algorithms that iterate over thousands of circuit generations and modify netlists by mutation operators. This approach often leads to unusual yet efficient AxCs as, for example, shown by the adders and multipliers provided in the EvoApproxLib [62].

Venkataramani et al. proposed SASIMI [94], which uses a substitute-and-simplify approximation technique. SASIMI identifies near-identical signal pairs, i.e., two signals which show similar behavior, and substitutes one with the other to simplify the logic. First, SASIMI evaluates and ranks signal pairs with a heuristic function including area and delay parameters. Then, with a gradient ascent technique, more accurately hill climbing, the highest-ranked pair is selected for substitution. The process is iterated until the user-defined quality constraints are violated. Additionally, the authors suggest the concept of *quality configurable circuits*, which are circuits that can operate in either an accurate or approximate version.

The SCALS framework presented by Liu and Zhang [50] initially maps a gate level logic network to a target technology. In an iterative process, SCALS extracts sub-netlists from the mapped netlist to which randomly chosen approximations or optimizations are applied. The candidates are then evaluated by a function including the area and the error, gained through a testing approach. A Metropolis-Hastings algorithm steers the candidate selection and the search until reaching a predefined number of iterations. Additionally, the user can specify a confidence interval for the estimated error. To evaluate the confidence on the estimated error, SCALS employs the T-test [99].

2.2 QUALITY ASSURANCE

Approximations applied to a design's candidates cause errors that propagate and potentially amplify or attenuate. In fact, the correlation between the errors at the candidates and the circuit's primary outputs is generally complex, making it a complex task to determine the resulting error at the circuit's primary outputs. Thus, approximate logic synthesis must employ a quality assurance step that checks whether an AxC satisfies the user-defined quality

constraints. The quality constraints are defined either by application-specific metrics, e.g., structural similarity [96], or by general error metrics, which can be further distinguished into *worst-case* error metrics (also denoted as *non-statistical* error metrics) or *average-case* error metrics (also denoted as *statistical* error metrics). This dissertation focuses on the general error metrics and discusses commonly used metrics in the following.

Consider the Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ with n primary input bits and m primary output bits that describes the correct functionality of circuit C, and let the Boolean function $\tilde{f} : \mathbb{B}^n \to \mathbb{B}^m$ of the AxC $\tilde{C}$ be an approximate function of f. The function $int(x) : \mathbb{B}^m \to \mathbb{Z}$ translates the binary vector x to an integer value, e.g., considering the binary representation $int(x) = \sum_{i=0}^{m-1} 2^i x_i$ (cf. [89]).

With the notation above, we can then define the following worst-case or non-statistical error metrics:

- *Worst-case absolute error* ϵ_{WC}: The worst-case (WC) absolute error (short: worst-case error), which is sometimes also denoted as *error magnitude* or *error significance* [89], computes the maximum absolute difference between the outputs of the original circuit C and the AxC $\tilde{C}$ over all input values. In fact, the WC error is a fundamental error metric [89] and is mainly considered in this dissertation. The WC error is defined as follows:

$$\epsilon_{WC}(f, \tilde{f}) = \max_{\forall x \in \mathbb{B}^n} \left| int(f(x)) - int(\tilde{f}(x)) \right| \tag{2.1}$$

- In its normalized form, we normalize the worst-case error ϵ_{WC} to the maximum output value of $f(x)$:

$$\epsilon_{WC,norm}(f, \tilde{f}) = \frac{\epsilon_{WC}(f, \tilde{f})}{\max\limits_{\forall x \in \mathbb{B}^n} int(f(x))} \tag{2.2}$$

- *Worst-case relative error* $\epsilon_{WC,rel}$: The worst-case relative error is similar to the worst-case error but computes the maximum relative difference between the original and the approximate outputs rather than the absolute difference. The worst-case relative error is defined as follows, where the division by $max(1, f(x))$ prevents a division by zero:

$$\epsilon_{WC,rel}(f, \tilde{f}) = \max_{\forall x \in \mathbb{B}^n} \frac{\left| int(f(x)) - int(\tilde{f}(x)) \right|}{max(1, f(x))} \tag{2.3}$$

- *Bit-flip error* ϵ_{BF}: The bit-flip error, also denoted as *maximum Hamming distance* [89], computes the maximum number of bits that discern between the precise value of f and the approximate value of $\tilde{f}$. The bit-flip error is defined as follows:

$$\epsilon_{BF}(f, \tilde{f}) = \max_{\forall x \in \mathbb{B}^n} \left(\sum_{i=0}^{m-1} f_i(x) \oplus \tilde{f}_i(x) \right) \tag{2.4}$$

The average-case or statistical error metrics are defined as follows:

- *Mean absolute error* ϵ_{MAE}: The mean absolute error (or *average-case arithmetic error*) sums the absolute differences between the original and approximated function and averages the results over the number of inputs:

$$\epsilon_{MAE}(f, \tilde{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \left| int(f(x)) - int(\tilde{f}(x)) \right| \tag{2.5}$$

- *Mean squared error* ϵ_{MSE}: The mean squared error is similarly computed as the mean absolute error; instead of computing the absolute difference, the difference between the original and the approximate function output is squared:

$$\epsilon_{MSE}(f, \tilde{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \left(int(f(x)) - int(\tilde{f}(x)) \right)^2 \tag{2.6}$$

- *Mean relative error* ϵ_{MRE}: The mean relative error sums the relative errors between the original and the approximate function and averages the result over the number of inputs:

$$\epsilon_{MRE}(f, \tilde{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \frac{\left| int(f(x)) - int(\tilde{f}(x)) \right|}{max(1, int(f(x)))} \tag{2.7}$$

- *Error rate* ϵ_{ER}: The error rate (or *error probability*) specifies the percentage of inputs for which the outputs of the original and the approximate function differ:

$$\epsilon_{ER}(f, \tilde{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} f(x) \neq \tilde{f}(x) \tag{2.8}$$

In order to assess the error or quality of an AxC, analytical methods [74, 76, 77, 80], testing-based approaches [65, 94], or formal verification [23, 69, 89–91] can be employed. Analytical methods employ error models for the input design, which allow to predict or estimate an AxC's error prior to its generation [5, 80]. While analytical methods can perform the predictions efficiently, the error model must be accurate and fully specified for the input design to deliver meaningful estimations. In fact, analytical methods are usually utilized to provide information to the ALS process for guidance rather than to verify an AxC's adherence to the quality constraints. Thus, the existing analytical approaches nevertheless rely on formal verification or testing in an additional quality assurance step to ensure the quality of a generated AxC [76, 77]. Furthermore, Vašíček observed that constructing accurate yet

simple mathematical models on the gate level is currently impossible, as described in Section 2.1.1.

Testing represents the most general approach for assessing the quality as the approach relies on circuit simulation, which works with any circuit design. In testing, a circuit simulator simulates the circuit using test vectors from a test vector set. For exhaustive simulation, the test vector set contains all possible inputs, meaning that all inputs are explicitly enumerated, which leads to an exponential worst-case execution time [90] and limits the approach to small-scale circuits. Thus, in practice, many authors scale the simulation's workload over the size of the test vector set and only employ a subset of all inputs. The larger the test vector set is, the higher is the confidence in the estimated error value. However, the testing runtime increases with the increasing size of the set [89].

Furthermore, employing a subset of all inputs is only viable for average-case (or statistical) metrics since these metrics converge towards a particular value. For determining worst-case metrics, the analysis of a subset of test vectors is insufficient since there is generally no correlation between the errors of different inputs, and convergence towards a particular error value is not given. As a result, an input that is not part of the subset but produces a quality violating error may remain undetected, which may result in a catastrophic underestimation of the actual worst-case error. Thus, formal methods are generally used to verify worst-case error metrics. Formal methods can also asses statistical metrics, and Vašíček [90] discusses algorithms for determining these metrics for combinational circuits. However, the experimental results from small-scale circuits highlight the complexity of determining such error metrics with formal methods. In fact, since error information over all input vectors must be counted, the problem becomes very challenging [90], and Chandrasekharan et al. [23], furthermore, demonstrate that some average-case metrics cannot be determined efficiently for practical designs, e.g., the mean absolute error.

Formal methods provide a guarantee on the result, which is always exact since formal methods verify all inputs – but without referring to the explicit enumeration of all inputs [90]. In fact, in this dissertation, we mainly consider formal methods in combination with the commonly used concept of *approximation miters*. This dissertation, however, makes no contributions to the formal verification methods or engines themselves since these belong to a separate field of research with its own challenges. Instead, this dissertation utilizes formal methods as a tool to facilitate the verification of AxCs, and contributes appropriate and novel quality assurance environments in the form of the approximation miters. Thus, we only briefly elaborate on general approximation miters and the verification engines in the following, and provide more detailed discussions on the related topics at the appropriate locations of this dissertation, i.e., Section 3.4.5 and Chapters 5 to 7.

A common approach in formal verification is to construct an *approximation miter* [22, 23, 69, 90, 91] which comprises the original circuit, the AxC, and logic for error computation. Then, instead of computing the exact error

value, which states an extremely hard task as described above, a satisfiability (SAT) problem is formulated. In essence, solving the SAT problem then answers whether the encoded logical formula can be satisfied. In the context of this dissertation, the question asked is whether the error between the original function f and the approximate function $\tilde{f}$ can exceed the user-defined threshold T: $\epsilon(f, \tilde{f}) \leqslant T$. In order to solve the SAT problem efficiently for combinational circuits, SAT solvers are being utilized, which operate on Boolean logic and can verify designs with millions of gates in a reasonable time [89]. The SAT solver verifies all possible inputs and reports *unsatisfiable* if no input assignment results in a violation of the error threshold T. In this case, the user holds a guarantee that the verified AxC is valid, i.e., cannot violate the error threshold (or quality constraints). If, however, the SAT solver finds an input assignment that results in a threshold violation, the AxC is invalid, and the solver reports *satisfiable* and returns the input assignment that caused the violation.

Besides SAT solvers, there are satisfiability modulo theories (SMT) solvers which generalize the SAT problem and combine a SAT solver with one or more theory solvers. In this way, the SMT solver operates at a higher level of abstraction [12] than a SAT solver that operates on Boolean logic only. In fact, SMT solvers verify the satisfiability of a formula that may contain operations from various theories, e.g., bit vectors, arrays, or integer arithmetic, and use the highly-specialized theory solvers for efficient solving. Assume, for example, that the problem statement $a + b = b + a$ has to be verified, with a and b being integers. A SAT solver must translate the problem into a formula that expresses the addition in Boolean logic at bit level. In contrast, a SMT solver can employ a theory solver, e.g., for integer arithmetic, to verify the statement at a higher level of abstraction and efficiently find the answer to the obvious problem statement.

In order to verify sequential circuits, input *sequences* have to be verified instead of individual input assignments. One approach to verify a finite sequence of a sequential circuit is bounded model checking, where a SAT solver verifies the unrolled circuit [23, 69]. Other approaches are inductive reasoning [120] or, more recently, property-directed reachability (PDR) [31], which facilitate the verification of an infinite number of time frames [23].

3

CIRCA: A SEARCH-BASED APPROXIMATE LOGIC SYNTHESIS FRAMEWORK

3.1 INTRODUCTION

This chapter presents our framework for search-based approximate logic synthesis (ALS), CIRCA [101]. CIRCA's main contribution is providing the research community of approximate computing (AC) with a framework that allows for the rapid implementation of new methods and establishes the setup of custom ALS processes at any of the levels of abstraction described in Section 2.1. In this way, CIRCA provides an ALS environment to fairly compare different methods and techniques under consistent conditions.

In Section 2.1.1, we have defined a general search-based ALS process with the main steps *search*, *approximate*, *verify*, and *estimate*. Consequently, the main task for an ALS framework is to implement these steps to enable a complete ALS process. In addition to this main task, we have found that a comprehensive ALS framework needs to be *general*, *modular*, *compatible*, *extensible*, and *available* (see Section 3.3). Our analysis of the related work (see Section 3.2), however, showed that existing ALS frameworks lack these key requirements and, instead, often implement the ALS process in monolithic blocks with interwoven phases. Due to this fact, a fair comparison of ALS methods is often not possible since the ALS setup may be inconsistent. Hence, in contrast to the existing frameworks, CIRCA considers these criteria.

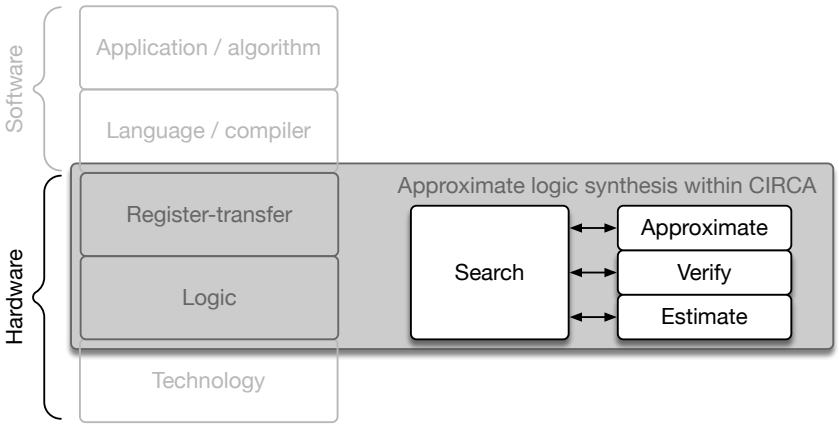


Figure 3.1: Approximate logic synthesis in the computing stack and CIRCA.

CIRCA provides an architecture that divides the ALS process into the four main steps as shown in Figure 3.1 and, from a high-level view, allocates an individual module for each step. The individual modules can be configured independently to facilitate swift changes in the ALS setup and even allow for complementing different methods. In this way, CIRCA enables the evaluation of new methods and fosters comparative studies by enabling a fair experimental comparison of alternative methods. For seamless integration into existing synthesis flows, separate input and output stages frame CIRCA's ALS process for pre-processing or post-processing, respectively.

In summary, we make the following contributions:

- We present CIRCA[1], a general, modular, compatible, and extensible open-source ALS framework, which the research community can easily employ to implement, verify, and evaluate emerging ALS methods.
- We demonstrate CIRCA's capabilities in our experiments by comparing three different search methods in combination with two different approximation techniques.

Multiple people contributed to CIRCA's development: Tobias Wiersema and Hassan Mohammadi Ghasemzadeh contributed to the conceptual design. Muhammad Awais provided CIRCA's Monte Carlo tree search implementation. My student research assistant, Matthias Artmann, was involved in the architectural development and the implementation of the framework. My contribution to this work lies in the conceptual and architectural design as well as in the implementation of CIRCA. In fact, I have initially started CIRCA's development in my Master's thesis [100], which Tobias Wiersema supervised. The outcome of the thesis has then been developed further to the state presented in this dissertation.

In Section 2.1 of the background chapter, we have discussed the state-of-the-art ALS frameworks related to CIRCA. To discuss CIRCA in detail, we structure this chapter as follows: For related frameworks, we first provide an evaluation in Section 3.2 and then identify key requirements for a flexible ALS framework in Section 3.3. Then, in Section 3.4, we present the CIRCA framework in detail, which was developed under the consideration of the identified key requirements. Finally, we demonstrate CIRCA's capabilities through experiments presented in Section 3.5 and conclude the chapter in Section 3.6.

We have previously presented the CIRCA framework at a workshop [103] and published CIRCA in a journal [101]. This chapter largely follows the discussion of the journal publication and cites most parts of the discussion of the classification of the related work (Section 3.2), the discussion on requirements for a comprehensive ALS framework (Section 3.3), and parts of the experimental results (Section 3.5). Section 3.4.5 combines discussions from [101, 105]. My colleague, Tobias Wiersema, contributed to the publication [105] and follows the discussions in his dissertation; hence parts of the

1 https://git.uni-paderborn.de/circa/public/circa_v2

discussions in Section 3.4.5 overlap in his and my dissertation. The distinct contributions, however, are made clear in Chapter 6.

3.2 CLASSIFICATION OF EXISTING FRAMEWORKS

Table 3.1 presents our attempt to classify a representative selection of ALS methods and frameworks from Section 2.1. From the classification, we then identify key requirements for a general ALS framework in Section 3.3. The challenges are two-fold. Firstly, we need to identify meaningful and orthogonal categories to provide a clear and distinct classification. Secondly, we have to retrieve the required information from related works. Table 3.1 states the classification of the ALS method according to the categories from Section 2.1 and comprises categories in four groups: the input, the ALS, the output, and whether the framework has been made publicly available.

Regarding the input, the first category is the circuit type. Most of the frameworks approximate combinational circuits, while ASLAN was developed for sequential circuits. However, it has to be noted that it is not necessarily the characteristic of the input circuit that determines its type in our classification. Rather, an approximation technique for sequential circuits means that at least the approximation technique or the quality assurance step (as for ASLAN) are considering the clocked nature of the circuit. For example, several frameworks use sequential circuits such as FIR filters as benchmarks but restrict the approximation to the datapath and do not report on testing the resulting circuit for a sequence of clock cycles or formally verifying it. Hence, we classify these approaches as *combinational* in Table 3.1. For ABACUS, we are somewhat uncertain how to classify it since the mentioned approximation techniques are clearly for sequential behavior, e.g., loop transformations, but a corresponding quality assurance was not detailed.

In terms of input model, SASIMI, SALSA, and AIG rewriting rely on gate level netlists or a Boolean representation. SCALS takes technology-mapped netlists as input, ASLAN begins with circuits described in a structural hardware description language (HDL), and ABACUS, operating on a more abstract level, requires a behavioral HDL or register-transfer level code. Another issue is how the user can control the error. Most frameworks allow for specifying an error bound, often in several error metrics such as error rate or mean absolute error. SALSA and ASLAN define quality functions or quality evaluation circuits, respectively, which encode error bounds. Again, ABACUS is different as it generates a Pareto front showing reasonable trade-offs between accuracy and power. A user-specified number of iterations controls the ALS process. Generally, more iterations lead to more approximations and, in turn, can add non-dominated designs to the Pareto front with larger errors.

The second group of rows in Table 3.1 characterizes the ALS, split into the three categories search, approximation technique, and quality assurance. All techniques rely on heuristics for search. The only exception is SALSA that does not apply a search method but systematically iterates over the outputs of the approximate circuit (AxC) to apply don't care optimization. Consequently,

Table 3.1: Overview over presented frameworks for approximate circuit generation. Adapted from Witschen et al. [101]

Category	**SASIMI** [94]	**SALSA** [91]	**AIG rewriting** [22]	**ABACUS** [64, 65]	**SCALS** [50]	**ASLAN** [69]
ALS method	Netlist transformation	Boolean rewriting	Boolean rewriting	AHLS	Netlist transformation	AHLS
Circuit type	Combinational	Combinational	Combinational	Combinational + sequential (?)	Combinational	Sequential
Input model	Gate netlist	Gate netlist	Gate netlist/AIG	Behavioral HDL	Gate/LUT netlist	Structural HDL + annotations
Error control	Error bound	Quality function	Error bound	#Iterations	Error bound	Quality evaluation circuit
Search method	Heuristic (hill climbing)	–	Heuristic (greedy)	Heuristic (greedy)	Heuristic (Metropolis-Hastings)	Heuristic (hill climbing)
AC technique	Substitute-and-simplify	Approx. don't care	AIG rewriting	AST transforms	Logic transforms	Precision scaling
Quality assurance	Testing	By construction	Formal verification	Testing	Testing	Formal verification
Output	Approx. circuit	Approx. circuit	Approx. circuit	Pareto front	Approx. circuit	Approx. circuit
Output model	Gate netlist	Gate netlist	Gate netlist (AIG)	Behavioral HDL	Gate/LUT netlist	Structural HDL
Target technology	Standard cell	Standard cell	Techn. independent	Standard cell	Std. cell/LUT-based	Standard cell
Publicly available	–	–	Yes	Yes	–	–

SALSA creates circuits that adhere to the error bound by construction, making a subsequent quality assurance step obsolete. ASLAN and AIG rewriting formally verify the AxC's quality, which is time-consuming but provides a much stronger statement about quality than the testing approaches used in SASIMI, ABACUS, and SCALS.

The next group of categories characterizes the result produced by the respective framework. Mostly, the tools return one AxC in the form of a gate level netlist or structural HDL. ABACUS, however, returns a set of designs in behavioral HDL at register-transfer level that form a Pareto frontier with respect to accuracy and power. Since the results of all frameworks are either netlists or synthesizable hardware descriptions, they can potentially target standard cell and FPGA technology. In contrast, with the category *target technology*, we refer to the technology used to get estimates for area, delay, and power during the synthesis process. Here, most frameworks target standard cell libraries except for AIG rewriting, where the and-inverter-graph (AIG) representation and ABC functions, respectively, are employed to retrieve technology-independent estimates for area and delay.

Finally, only the authors of ABACUS and AIG rewriting decided to make their frameworks publicly available.

3.3 REQUIREMENTS FOR A FLEXIBLE FRAMEWORK

Our analysis of related frameworks and the attempt to categorize them has shown that all these approaches have been developed for specific circuit types and limited approximation techniques. In particular, ALS is typically described as a monolithic block with interwoven phases for approximation, search, and assuring quality. Moreover, only few frameworks are openly available for experimentation. This situation severely hampers the development and evaluation of new techniques for ALS, and the comparison to existing ones.

With CIRCA, we aim at overcoming these shortcomings and provide a flexible framework for ALS. As a starting point for this development, we take our classification of Table 3.1. This classification provides several categories and shows that many of these are largely orthogonal, giving rise to a reasonable structuring of our ALS framework. Generally, we envision a framework that fulfills the following technical key requirements:

- *General*: The framework should not be restricted to certain circuit types, error metrics, approximation and search techniques, or specific target technologies.
- *Modular*: The framework architecture should enable the exchange of certain processing steps without affecting other steps. Modularity is key for the evaluation and the comparison of different techniques under a consistent experimental setup.
- *Compatible*: The framework, in particular its inputs and outputs, should connect to other, widely-used academic and commercial front-end

and back-end tools, e.g., tools synthesizing circuits for ASIC or FPGA technology.

- *Extensible*: The framework should facilitate the swift implementation and evaluation of new techniques.

Additionally, the framework should satisfy the community requirement:

- *Open source availability*: The framework should be publicly available and allow other researchers to use, modify, and extend it. Open-source availability encourages the evaluation and comparison of new techniques against existing ones, and fosters comparative studies.

3.4 THE CIRCA FRAMEWORK

Considering the previous discussion on related work and identification of key requirements, we present our CIRCA framework in the following sections as a flexible ALS framework. First, we will describe CIRCA's general concept. Then, we will detail on CIRCA's individual building blocks and underlying principles. Finally, we classify CIRCA into the categories from Section 3.2.

3.4.1 *The Concept of CIRCA*

Figure 3.2 shows CIRCA's conceptual design, which has been developed under the consideration of the key requirements of an ALS framework (general, modular, compatible, extensible, and open-source, cf. Section 3.3) and divides into three stages: the *input* stage, the *QUAES* stage, and the *output* stage. While the quality assurance, approximation, estimation, and search space exploration (QUAES) stage implements the main ALS process, the input and output stage frame the QUAES stage and pre-process the input design from preceding tools or post-process the output designs for succeeding tools, respectively. Via a separate interface, CIRCA enables communication with external tools, e.g., ABC [14], Yosys [110], or Xilinx Vivado.

The input stage fulfills the two main tasks of pre-processing the input design and ensuring compatibility between CIRCA and external tools and formats. In the pre-processing, the input stage has to identify the candidates in the input design. Candidates are usually arithmetic components from a design's data path, e.g., adders or multipliers, and can be identified either through automated methods or manually by the user through code annotations in the input design. In order to ensure CIRCA's compatibility, the input stage has to translate the input design into one of the formats CIRCA uses internally, Verilog or Berkeley logic interchange format (BLIF).

The QUAES stage implements the ALS process and further subdivides into the processing blocks *quality assurance*, *approximation*, *estimation*, and *search space exploration*, and the blocks perform operations following their descriptive names.

CIRCA targets search-based ALS processes as described in Section 2.1, and the search space exploration block implements the algorithm that explores

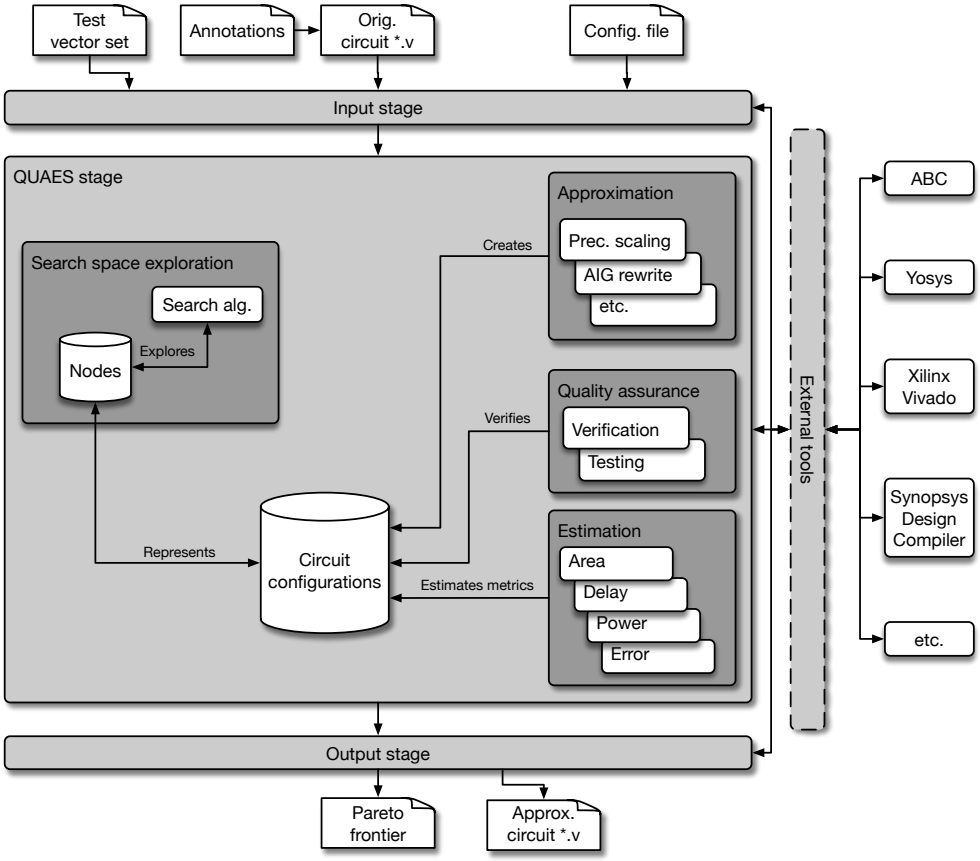


Figure 3.2: Overview of CIRCA's conceptual design. Extended from [101].

the search space and acts as the central control block of the QUAES stage. The search algorithm operates on *nodes*, i.e., abstract representations of AxCs that dissociate circuit-specific information, and CIRCA relates a node to a unique AxC outside of the search. In this way, the search is kept agnostic of the context of AxCs, which allows the utilization of a wide range of off-the-shelf search algorithms, e.g., hill climbing or breadth-first search, and renders domain-specific adjustments to the search algorithm obsolete. Since the configuration of the candidates, i.e., the approximation technique and the specific parametrization of the individual candidates, uniquely describes an AxC, CIRCA refers to AxCs also as *circuit configurations*.

Throughout execution, the search explores nodes and queries information on different properties, i.e., validity and performance in a target metric. In order to determine the different properties, CIRCA automatically translates a node to its corresponding AxC and invokes the required circuit-specific operations. If the AxC has not been generated yet, CIRCA firstly invokes the approximation block for AxC generation.

The approximation block approximates the candidates using the approximation technique and the parameters encoded in the node or circuit configuration, respectively. CIRCA then uses the approximated candidates to assemble the AxC by replacing the candidates in the input design with their approximated counterparts. Afterward, CIRCA invokes the corresponding

processing block on the circuit configuration to provide the information requested by the search.

If the search requests the node's validity, CIRCA invokes the quality assurance block. The quality assurance block verifies whether the AxC adheres to the user-specified quality constraints through formal verification, testing-based approaches, or analytical methods. If the AxC satisfies the constraints, the AxC and the corresponding node are considered valid; otherwise, both are invalid.

If the search requests the node's performance, CIRCA invokes the estimation block to determine the target metrics, usually through external tools, e.g., ABC [14] or Yosys [110]. The estimation block can directly attribute the AxC with the target metrics, e.g., hardware area, delay, power consumption, or the AxC's error, or implement a heuristic function to aggregate multiple performance metrics into a single value.

When the search terminates and thus completes the ALS, the output stage is invoked for post-processing. Post-processing involves selecting and preparing AxCs for succeeding back-end tools for actual circuit implementation. First, the output stage receives the search's result, which may be multiple nodes or a single node, and translates them to AxCs in the desired output format. Then, the output stage selects as CIRCA's output a single AxC, e.g., the circuit that optimizes the target metric, or the stage selects a set of AxCs as CIRCA's output, e.g., the Pareto frontier.

CIRCA's ALS process is fully configurable by the user through a *configuration file*. In the file, the user can set up each stage, block, and method individually. In addition, if the candidates are known in advance, the user can provide tailored configurations to customize operations on specific candidates, e.g., to tailor an approximation technique towards a specific candidate. If not specified otherwise, CIRCA refers to the default configuration specified by the user.

3.4.2 *Search Space Exploration*

Figure 3.3 shows the search space exploration block in more detail. In the initial publication of CIRCA [101], the search space exploration block suggested a procedure that is followed by many search algorithms: 1) select a node, 2) expand the node for search space exploration, and 3) evaluate newly explored nodes. The procedure, however, may impose artificial restrictions upon the search algorithm. Thus, the design of the search space exploration block was adjusted; the block now further divides into different independent modules, each fulfilling a specific task to provide a modular and generic architecture.

As described in the previous section, the *search* block implements the search algorithm, which operates on nodes only. A node is an abstract data structure and attributes as properties a node's validity, performance, and relationships, such as parents, children, and siblings. The relationships of the nodes define a search graph, and the search graph represents the search space

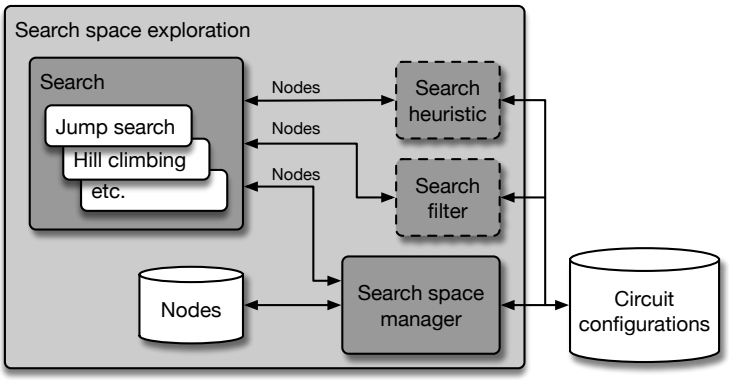


Figure 3.3: CIRCA's search space exploration block in more detail. Dashed lines indicate optional blocks.

that the search algorithm explores. Each node explored can then be evaluated based on its validity and performance – for which CIRCA automatically invokes the required circuit-specific operations, as described in Section 3.4.1. The search is thus agnostic and abstracted from the context of circuits and ALS, which enables a straightforward implementation of a wide range of off-the-shelf search algorithms. Currently, CIRCA implements the following search algorithms:

- the local searches simulated annealing and hill climbing (see [101] for implementation details),
- a breadth-first search,
- the genetic algorithm non-dominated sorting genetic algorithm (NSGA-II) [28],
- a heuristic Monte-Carlo tree search (see [101] for implementation details),
- the domain knowledge exploiting jump search (see Chapter 4).

The blocks *search filter* and *search heuristic* are optional for an ALS process and enable the search to exploit domain knowledge to improve the ALS' performance. In contrast to the search, both blocks have access to circuit-specific properties of a node, i.e., the blocks operate on nodes and the corresponding circuit configurations. In this way, the search algorithm itself is kept general by abstracting domain-specific knowledge, yet domain knowledge can be exploited during the search if desired.

The input to the filter is a list of nodes to which the block applies a filter function that considers circuit-specific properties or domain knowledge to remove unfit nodes. A filter can, for example, be used to enforce constraints on circuit-specific parameters, such as delay or hardware area, or to compare nodes against each other. The search filters currently implemented in CIRCA are:

- a random filter that filters randomly selected nodes, and

- a k-means clustering [67] filter that clusters nodes based on the errors of the candidates and only returns the centroids of the clusters.

The search heuristic can be used to evaluate nodes with a heuristic function that considers circuit-specific properties, e.g., error information of candidates as done in jump search (see Chapter 4). While the estimation block in the QUAES stage can also implement a heuristic function, the search heuristic is more specific since the block can also consider a node's relations and circuit-specific properties. Currently, CIRCA implements the following search heuristics:

- a method showcasing the block's functionality by evaluating a node in terms of delay or hardware area, and
- a method to compute jump search's heuristic function (see Chapter 4).

The *search space manager* creates the nodes and arranges them in the search space by defining their relationships, i.e., the parents, children, and siblings of a node. In this way, CIRCA abstracts circuits to nodes and separates the construction of the search space from the search algorithm, which eases the implementation of new search algorithms and enables greater flexibility, since a swift change of the search space can be performed without requiring changes in other parts of the search space exploration block.

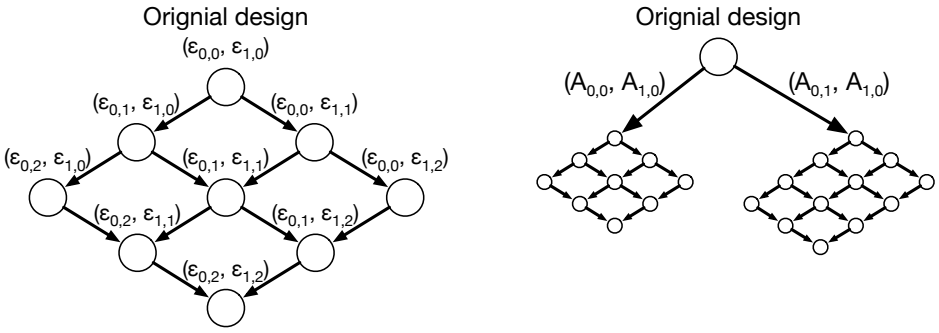
Many factors qualify for defining the relationships, and the construction of the search space can become arbitrarily complex. Thus, the search space manager allows implementing custom algorithms to define the relationships between nodes. Furthermore, as the relationships usually depend on the approximation techniques and their parameters as well as circuit-specific properties, e.g., the quality of the candidates, the search space manager operates on nodes and the corresponding circuit configurations.

CIRCA implements two search space managers. A general search space manager that defines the parents/children of a node by the quality of the candidates so that a parent/child instantiates one candidate with the next-highest/next-lowest quality. The second manager builds upon the general manager but exploits information from the method described in Chapter 7 to prune nodes from the search space to reduce the search space's size.

Figure 3.4 visualizes two exemplary search spaces constructed with the general search space manager for a design with two candidates. In Figure 3.4a, one approximation technique per candidate is employed, and $\epsilon_{i,j}$ is used to describe the candidates' local quality constraints (LQCs), i.e., quality constraints that are imposed upon and limited to the candidate. The first index i of ϵ indicates the candidate; the second index j indicates the candidate's LQC, which is ranked in ascending order, i.e., LQC $\epsilon_{i,l}$ imposes stronger error bounds than LQC $\epsilon_{i,m}$ for $i \in \{0,1\}, l < m$. A tuple $(\epsilon_{0,j}, \epsilon_{1,j})$ comprises the LQCs of the candidates and defines each node in the example. The tuple $(\epsilon_{0,0}, \epsilon_{1,0})$ represents the original design since both candidates are assumed error-free.

As described above, the general manager defines those nodes as a current node's children that instantiate one candidate of the next-lower quality. For

example, in Figure 3.4a, the children of the original design are the nodes $(\epsilon_{0,1}, \epsilon_{1,0})$ and $(\epsilon_{0,0}, \epsilon_{1,1})$. The resulting search space then has a shape of a lattice.



(a) A search space for two candidates with one approximation technique each.

(b) A search space for two candidates, one employing two approximation techniques.

Figure 3.4: Exemplary search spaces as constructed by the general search space manager.

Figure 3.4b visualizes the more complex scenario where a candidate employs more than one approximation technique. In the example, candidate 0 employs the approximation techniques $A_{0,0}$ and $A_{0,1}$, while candidate 1 only employs approximation technique $A_{1,0}$. The general search space manager then constructs a search space that considers the possible combinations of approximation techniques and LQCs. As shown in Figure 3.4b, the manager constructs lattice-shaped sub-spaces for the specific combination of approximation techniques and the sub-spaces contain the LQC combinations of the candidates. As a result, the search space expands in a controlled manner from the least to the most aggressively approximated nodes. The resulting size of the search space then equals to the sum of the sizes of the individual sub-spaces whose size can be computed via the product of the LQCs of the candidates, using the following equation:

$$\sum_{a \in A} \prod_{c \in C} | \, \mathrm{LQCs}(c, a_c) \, | \tag{3.1}$$

The tuple $a \in A$ specifies a combination of approximation techniques and the set A holds all possible combinations. The set C contains all candidates, and the function $\mathrm{LQCs}(c, a_c)$ returns the set of distinct LQCs under the approximation technique a_c for candidate $c \in C$.

The example also shows that we can uniquely describe a node with the candidates' approximation techniques and LQCs. Thus, CIRCA encodes the information into two tuples to represent each node and circuit configuration uniquely in an abstract yet unified data structure. For example, the node described by the tuples $\{(A_{0,1}, A_{1,0}), (\epsilon_{0,2}, \epsilon_{1,1})\}$ could encode for candidate 0 approximation-aware AIG rewriting in maximum effort mode in $A_{0,1}$ and

$\epsilon_{WC} \leqslant 3$ as LQC in $\epsilon_{0,2}$. The approximation block decodes the values in the tuples and deduces the approximation technique and corresponding approximation parameters for each candidate (see Section 3.4.3) to invoke the appropriate techniques. Furthermore, as the possible parameters and the ordering of the LQCs is unique to and thus determined by the approximation techniques, the techniques provide this information to the circuit configuration, which the search space manager eventually utilizes to arrange the nodes in the search space.

At this point, we want to note that CIRCA does not expect the candidate's approximations to be ordered in terms of the LQCs, since the metric used for the ordering is abstracted through the encoded tuples. In fact, the ordering in the approximation block can be chosen freely, and any metric a designer deems suitable can be used. In this dissertation, however, all orderings base on the LQCs, allowing for a controlled search space expansion. From a technical point of view, the search space manager can change the ordering and prepare own orderings, which, however, results in a highly customized, non-generic search space that likely shows high dependencies between the different processing steps in CIRCA.

3.4.3 *Approximation*

The approximation block implements the approximation techniques to approximate the candidates. A setup phase attributes at least one approximation technique to each candidate and initially analyzes the candidate's information to set up the technique for each candidate individually. Setting up the approximation method depends on the respective approximation technique and, for example, demands determining compatible components from an approximate component library or the maximal number of bits that can be truncated from a candidate. During ALS, CIRCA automatically invokes the approximation block if new circuit configurations need to be generated.

Furthermore, each approximation technique of each candidate initially generates an order of the possible approximations and then provides that order to the search space manager, as mentioned in Section 3.4.2, to enable controlled and gradual advances in the approximations. The designer defines the order arbitrarily but could, for example, consider the degree of approximation that can be applied to the candidates with the respective technique.

In order to control the degree of approximation, approximation techniques usually provide control parameters, e.g., parameters to control the resulting quality in terms of error metrics via LQCs. As approximation techniques usually employ different parameters, CIRCA uses an encoded representation for the approximation techniques and their parameters to unify the information outside the approximation block. In this way, for example, the search space manager can directly utilize the encoded information in the ordering to define children, parents, and siblings. To nevertheless enable highly customized approaches, the approximation block can be queried to

decode the information and provide the raw information, which can then be processed, e.g., in the search space manager to construct tailored search spaces or in a search heuristic.

CIRCA currently implements as approximation techniques:

- precision scaling,
- approximation-aware AIG rewriting [22], and
- the EvoApproxLib [62].

3.4.4 *Estimation*

CIRCA invokes the estimation block to estimate the target metrics for a circuit configuration. The metrics can either be estimated via internal functions or external tools, e.g., ABC [14] or Xilinx Vivado, accessed through CIRCA's *external tool* interface. The estimation block is kept generic, and any target technology or metric is supported. CIRCA can currently estimate the following performance metrics of circuits:

- hardware area and delay for standard cell technology using ABC [14], Synopsys Design Compiler, or Yosys [110],
- hardware area and delay for field-programmable gate array (FPGA) technology using ABC [14], Yosys [110], or Xilinx Vivado, and
- power consumption using Xilinx Vivado or Synopsys Design Compiler.

The determined metric values can then be directly attributed to the circuit configuration or node, respectively, or a heuristic function aggregates multiple values into a single value.

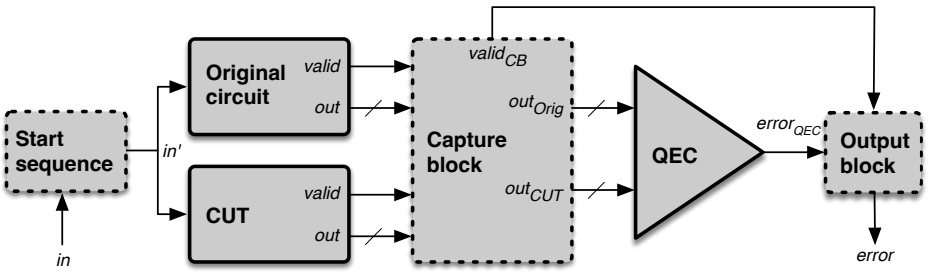
3.4.5 *Quality Assurance*

The quality assurance block verifies whether an AxC satisfies the user-defined quality constraints and refers to the approximated yet not verified circuit as circuit-under-test (CUT). For quality assurance, CIRCA can employ testing or formal verification. While approaches based on formal verification lead to conceptually much stronger statements about quality than testing, they also tend to endure very long runtimes.

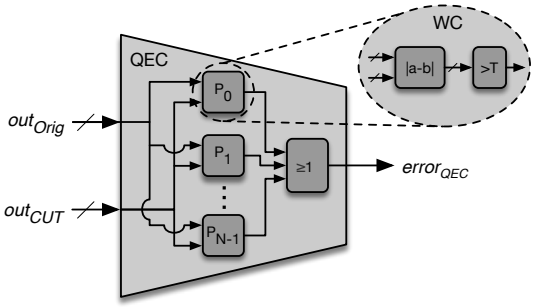
For testing, either the user provides a set of test vectors or CIRCA generates a random test vector set. Quality assurance procedures can then utilize a circuit simulator and apply all of these vectors or a randomly selected subset to a CUT, i.e., the AxC whose quality is checked. Comparing the output of the CUT against the precise output data then allows for computing the quality. As an exhaustive simulation of all input vectors is generally not feasible, testing is usually employed for statistical error metrics such as the mean squared error or the error rate.

Formal verification focuses on the property *equivalence* between a circuit specification and its implementation. In the context of AC, however, this

property needs to be relaxed to *equivalence up to some bound* [89]. We use an approximation miter to verify this property, as shown in Figure 3.5a. Following the terminology of Ranjan et al. [69], we denote the approximation miter as sequential quality constraint circuit (SQCC).



(a) Sequential quality constraint circuit.



(b) Quality evaluation circuit.

Figure 3.5: Overview of the sequential quality constraint circuit and the quality evaluation circuit. Taken from [101].

As Figure 3.5a outlines, the SQCC comprises the original circuit, the CUT, and the quality evaluation circuit (QEC) that forms a property checker. The three configurable blocks — depicted by dashed boxes — extend the applicability of our verification approach to a large range of circuits. The block *start sequence* primes the original circuit and the CUT for operation by implementing a startup protocol. A typical startup protocol might first assert a reset signal and then a start or enable signal before beginning the actual computation on the input sequence. This additional knowledge may decrease the runtime of the verification since fewer state transitions have to be considered. The *capture* block and *output* block work together to ensure that error bounds are checked only when the original circuit and the CUT are specified to generate valid output signals.

When inputs are applied to purely combinational circuits, the result is immediately present at the primary outputs (POs). Thus, the SQCC's configurable blocks are simply passing through the signals for this circuit type. For sequential circuits, we distinguish between three different types, which require different configurations of the blocks capture and output (cf. Figure 3.6):

(i) Run-to-completion (RTC): This circuit type reads inputs and produces an output, whose presence is indicated by a valid flag. We allow different latencies between the original circuit and the AxCs. The SQCC ensures the

error bound verification is conducted at the correct clock cycles of the original circuit and the CUT, which is visualized in column *RTC* of Figure 3.6. The RTC circuit type has also been used by ASLAN [69].

(ii) Streaming (STR): The second circuit type also covers sequential circuits with valid flags. However, instead of only comparing one set of results at the end of the computation, we allow for an endless stream of results, each indicated by a raised valid flag. For this case, we require the valid flags of the original circuit and the CUT to be checked at the same clock cycles, i.e., both circuits must have the same latencies. This case is exemplified in column *STR* of Figure 3.6.

(iii) Cycle-by-cycle (CBC): Circuits of this type come without a valid flag indicating the completion of a result. Hence, we have to be more strict when verifying the error bounds and check for quality in every single clock cycle. The capture and output blocks of the SQCC are turned into pass-through circuits. Column *CBC* of Figure 3.6 shows this case. Although the CBC type can be seen as a special case of the STR type, we have to distinguish them in terms of automating the verification setup and forming the SQCC.

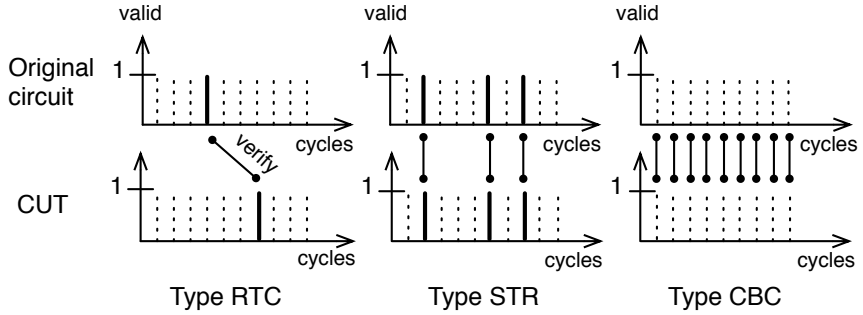


Figure 3.6: Circuit types with resulting verification steps.

The QEC generates a single output flag $error_{QEC}$, which is raised if the comparison between the output signals of the original circuit and the CUT indicates a quality constraint violation. Figure 3.5b displays the QEC, which holds one or more encoded quality constraints, $P_0, \ldots, P_{N-1}$. The corresponding quality constraint checker modules are OR-ed to form the output flag $error_{QEC}$. The figure furthermore details the encoding of a worst-case (WC) error bound in P_0. First, the absolute value of the difference of the two POs of the original circuit and the CUT is computed. Then, the result is compared against a specified threshold T. If all possible deviations are lower than this threshold, the error flag will never be raised, meaning that the quality constraints are never violated. Other error metrics can be encoded similarly. The bit flip error, for example, is determined by counting the number of differing bits in the two output patterns of the original circuit and the CUT. Note, however, that some error metrics cannot yet be efficiently verified using formal verification methods (see Section 6.4.1), e.g., statistical error metrics, such as the mean squared error [23, 90], or generally metrics that involve divisions [23].

Depending on the original circuit, the CUT, the error bounds, and the circuit type, we set up the SQCC. The verification task is then to prove that, for any input assignment or input sequence, the SQCC never reaches a state in which it raises its output `error`, and thus, indicates a quality constraint violation. Consequently, to formally verify that the quality constraints hold, we need to prove the unsatisfiability of the SQCC's output. CIRCA currently offers two different inductive solvers for this step from the tools ABC [14] and Yosys [110]. For combinational circuits, CIRCA additionally provides the satisfiability modulo theories (SMT) solver z3 [61].

For error metrics, we distinguish between statistical and non-statistical metrics. Since statistical error metrics cannot yet be efficiently verified using formal verification methods, CIRCA employs testing and currently implements one statistical error metric:

- the mean squared error.

For the following non-statistical error metrics, CIRCA utilizes formal verification:

- the worst-case error, and
- the bit-flip error.

3.4.6 *Classification of CIRCA*

Table 3.2 classifies CIRCA into the categories identified in Section 3.2. The table distinguishes between CIRCA's concept and the implemented functionality. As the table shows, CIRCA's current implementation provides different methods for the particular categories. Conceptually, however, CIRCA provides large flexibility to developers and a highly configurable system to users.

3.4.7 *The Configuration File*

The user prepares a configuration file that contains all information to set up a concrete ALS process, and CIRCA uses Python's *configparser* [47] to process the configuration file. The configuration file separates into individual sections to set up stages and processing blocks via parameter-value pairs. The parameter `ID` is mandatory for each stage or processing block, and CIRCA uses the parameter to instantiate the correct class object for the method to provide the desired functionality. In addition, method-dependent parameters might follow, and dedicated sections allow the user to configure candidates individually to customize the ALS process for the candidate's properties. An exemplary configuration file is described in the following, which includes all mandatory sections and some optional configuration possibilities to showcase CIRCA's flexibility.

Listing 3.1 shows mandatory sections to describe the input design and the user-defined quality constraints to CIRCA. In the configuration file,

Table 3.2: Overview and classification of CIRCA for approximate logic synthesis. Adapted from Witschen et al. [101]

Category	**CIRCA (concept)**	**CIRCA (current implementation)**
ALS method	Independent	AHLS with Boolean rewriting and netlist transformation for candidates
Circuit type	Combinational + sequential	Combinational + sequential
Input model	Configurable*	Verilog HDL
Error control	Configurable	Sequential quality constraint circuit[II] Error bound(s)
Search method	Configurable	Hill climbing Simulated annealing Monte Carlo tree search NSGA-II Jump search Breadth-first search
AC technique	Configurable	Precision scaling AIG rewriting EvoApproxLib
Quality assurance	Configurable	Formal verification Testing
Output	Approx. circuit(s)	Approx. circuit(s)
Output model	Configurable*	Verilog HDL Gate netlist
Target technology	Techn. independent	Techn. independent
Publicly available	Not applicable	Yes

* For example, SystemC, behavioral or structural HDL, gate netlist, AIG, etc.
II For combinational circuits, registers in the sequential quality constraint circuit are bypassed.

the section `OriginalCircuit` gathers information on the input design and contains two parameters: `Module`, pointing to the input design's information in section `Module:top_mod`, and `QAMethod`, pointing to the quality constraints and the method used for verification in section `QAMethod:qa_top`.

```
# General information about the input circuit
[OriginalCircuit]
Module = mod_top
QAMethod = qa_top

# Information about the top-level module 'top'
[Module:mod_top]
Name = top
CircuitType = run_to_completion
RelevantOutputs = data_out
Partition:data_out = [23:16],[15:8],[7:0]
ValidSignal = valid
ResetSignal = rst
HighActive:rst = True

# Quality assurance for the top-level module 'top'
[QAMethod:qa_top]
ID = abc_dprove
QC:WC = 42
```

Listing 3.1: Configuration of the input design.

The section `Module:mod_top` contains information on the top module of the input design, which is relevant for ALS, e.g., the top module's name, the circuit type, and the output that has to satisfy the user-defined quality constraints. Furthermore, the user can specify whether an output signal partitions into multiple output signals and provide information on commonly used circuit signals, e.g., reset signals and valid flags. In fact, sections with the prefix `Module:` contain specific information about modules of the input design and can also be used to, for example, provide information on candidates, which is described later using Listing 3.5.

The section `QAMethod:qa_top` configures the quality assurance block and defines the quality constraints imposed upon the relevant outputs of the top module, i.e., on the three partitions of *data_out*. In the example, ABC's [14] *dprove* command is employed via the parameter `ID` to verify the WC error with a bound of 42. Sections with the prefix `QAMethod:` specify quality constraints and methods for verifying the constraints. CIRCA allows the definition of multiple such sections, which are then assigned to either the input design or candidates using the section's name that follows the prefix, e.g., `qa_top`. In this way, CIRCA can verify combinations of different quality constraints and even invoke quality verification on individual candidates.

Listing 3.2 shows the sections `Evaluation` and `Estimation`, which jointly set up the estimation block in CIRCA. The section `Evaluation` specifies the target metric – hardware area in this case. The section `Estimation` specifies the employed back-end for synthesis – lookup table (LUT) mapping via ABC in this case.

```
# Evaluation and Estimation specify the target metric
# Estimation calls tools to acquire target metric information
# Evaluation may process information further, e.g.,
# by incorporating metric values into a heuristic function
[Evaluation]
ID = eval_area
[Estimation]
ID = abc_if
```

Listing 3.2: Configuration of the estimation block.

Listing 3.3 shows the ALS setup for the search space exploration block. The setup configures CIRCA to utilize jump search (see Chapter 4) as search method with the associated search heuristic that provides domain-specific knowledge for jump search in the section `SearchHeuristic:js_heuristic`. In addition, the selected search space manager `General` defines the relations between the nodes via the errors of the individual candidates (cf. Section 3.4.2).

```
# Specifying the search space manager
[SearchSpaceManager]
ID = General

# Setup for the search method
[Search]
ID = JUMP_SEARCH
Heuristics = js_heuristic

[SearchHeuristic:js_heuristic]
ID = ImpactFactorsAreaHeuristic
FoM = fom_c
ImpactFactorsErr = {'cand_cstm': 0.67, 'cand_0': 1.0}
ImpactFactorsArea = {'cand_cstm': 0.4, 'cand_0': 0.37}
```

Listing 3.3: Configuration of the search space exploration block.

Similar to quality assurance, multiple approximation techniques can be specified in the configuration file via sections with the prefix `Approximator:`, which are then attributed to candidates. The example in Listing 3.4 shows the specification of approximation-aware AIG rewriting [22] and precision scaling. In the section `Default`, precision scaling is then set as the default approximation technique for candidates that have not been customized by the user in other sections of the configuration file. Generally, the section `Default` defines the default or fallback values for the ALS process in case of missing information.

Listing 3.5 shows sections for customizing the candidate `cand_cstm`. Similar to section `OriginalCircuit`, the user provides information on the candidate's module in section `Module:cand_cstm`. In this case, only the module's name is provided, which is required for CIRCA to parse the design for the corresponding candidate module. CIRCA then automatically extracts the remaining information, e.g., the candidate's outputs. Furthermore, the candidate is assigned quality constraints and both approximation techniques specified in

```
# Setup for the approximation methods: precision saling and AIG rewriting
[Approximator:app_ps]
ID = PS
[Approximator:app_aig]
ID = AIG

# Setup for default methods
[DEFAULT]
Approximators = app_ps
```

Listing 3.4: Configuration of the approximation block and the default configurations.

Listing 3.4, approximation-aware AIG rewriting and precision scaling. Specifying the approximation techniques for the candidates overrides the default values from section `Default`, meaning that, different from the other candidates in the design, the candidate employs two approximation techniques. Using the specified method in section `QAMethod:qa_cand_cstm`, CIRCA performs a dedicated quality check of the candidate's quality constraints after each approximation of the candidate. Defining quality constraints for an individual candidate can be useful if, for example, approximation techniques are used that offer no native quality check for particular error metrics.

```
# Customizing the candidate 'cand_cstm'
[Candidate:cand_cstm]
Name = cand_cstm
Module = cand_cstm_mod
Approximators = app_ps, app_aig
QAMethod = qa_cand_cstm

[Module:cand_cstm_mod]
Name = cand_cstm

[QAMethod:qa_cand_cstm]
ID = abc_dprove
QC:BF = 4
```

Listing 3.5: Candidate-specific configurations.

3.5 EXPERIMENTAL RESULTS

In this section, we summarize the experimental results from the CIRCA's initial publication [101] to demonstrate CIRCA's generality and to motivate the further research topics of this dissertation.

3.5.1 *Experimental Setup*

For experimentation, we have selected seven sequential circuits from our open-source benchmark suite PaderBench[2] and manually annotated adder

2 http://go.upb.de/paderbench

and multiplier components in the data path as approximation candidates. Table 3.3 gives an overview of the benchmark circuits, showing the name of the benchmark, the primary output's bit width, the hardware area in terms of the number of 4-input LUTs reported by ABC's *if* command, and the number of candidates.

Table 3.3: Sequential benchmark circuits

Benchmark Name	Output Bit Width	#4-LUTs	#Candidates
butterfly [71]	32†	19,038	7
fir_8tap	67	12,401	15
fir_pipe_16 [53]	18	8935	23
pipeline_add [27]	40	572	2
rgb2ycbcr [52]	24‡	4981	5
ternary_sum_nine [27]	20	1484	4
weight_calculator	12	2272	4

† Concatenation of real and imaginary part.
‡ Concatenation of three channels, each 8-bit wide.

We vary the WC error bound from 0.50% to 5.0%, expressed in percentage of the circuit's maximum possible output value, to employ formal verification with ABC's [14] *dprove* for assuring quality and the hardware area as target metric. Moreover, we have systematically experimented with the three search methods hill climbing (HC), simulated annealing (SA), and Monte Carlo tree search (MCTS) in different parametrization, and with precision scaling (PS) and approximation-aware AIG rewriting (AIG rewriting) [22] as approximation techniques. While this dissertation recapitulates the most important findings, CIRCA's publication [101] gives a full, in-detail discussion.

We have run the ALS process five times for each benchmark circuit and determined the averages as representative results. The experiments have been performed on the OCuLUS compute cluster of the Paderborn Center for Parallel Computing (PC²), which runs Scientific Linux 7.2 (Nitrogen), and we have provided one core of an Intel® Xeon E5-2670@2.6GHz and 2 Gigabyte main memory for a single benchmark run.

3.5.2 *Experimental Evaluation*

Figure 3.7 shows an excerpt of the experimental results presented in CIRCA's publication [101] and highlights the most important findings for this dissertation. The figure divides into seven bar plots, one for each benchmark, and displays the area normalized to the area of the original circuit over the normalized WC error bound. Each bar plot shows results achieved with different setups regarding the search method and approximation technique; the ALS setup is denoted as `search-method_approx.-technique`.

Comparing the approximation techniques over the experiments, we see that PS could achieve area savings of up to ≈ 55% (cf. `pipeline_add` with SA), while AIG rewriting could only achieve area savings of up to ≈ 25%

(cf. `ternary_sum_nine` with SA). An explanation for the superiority of PS over AIG rewriting in our experiments is that we have selected arithmetic components as candidates. For such components, precision scaling degrades the accuracy more gracefully than AIG rewriting since AIG rewriting targets the critical path with the approximations, which usually affects the carry-chain of a multiplier or adder. Thus, the approximations may lead to excessive errors and, in turn, to rejecting the AxC if moderate error bounds are applied. On the other hand, PS introduces smaller errors when operating on the least significant bits of the output vector of a candidate and thus approaches the error boundaries more carefully.

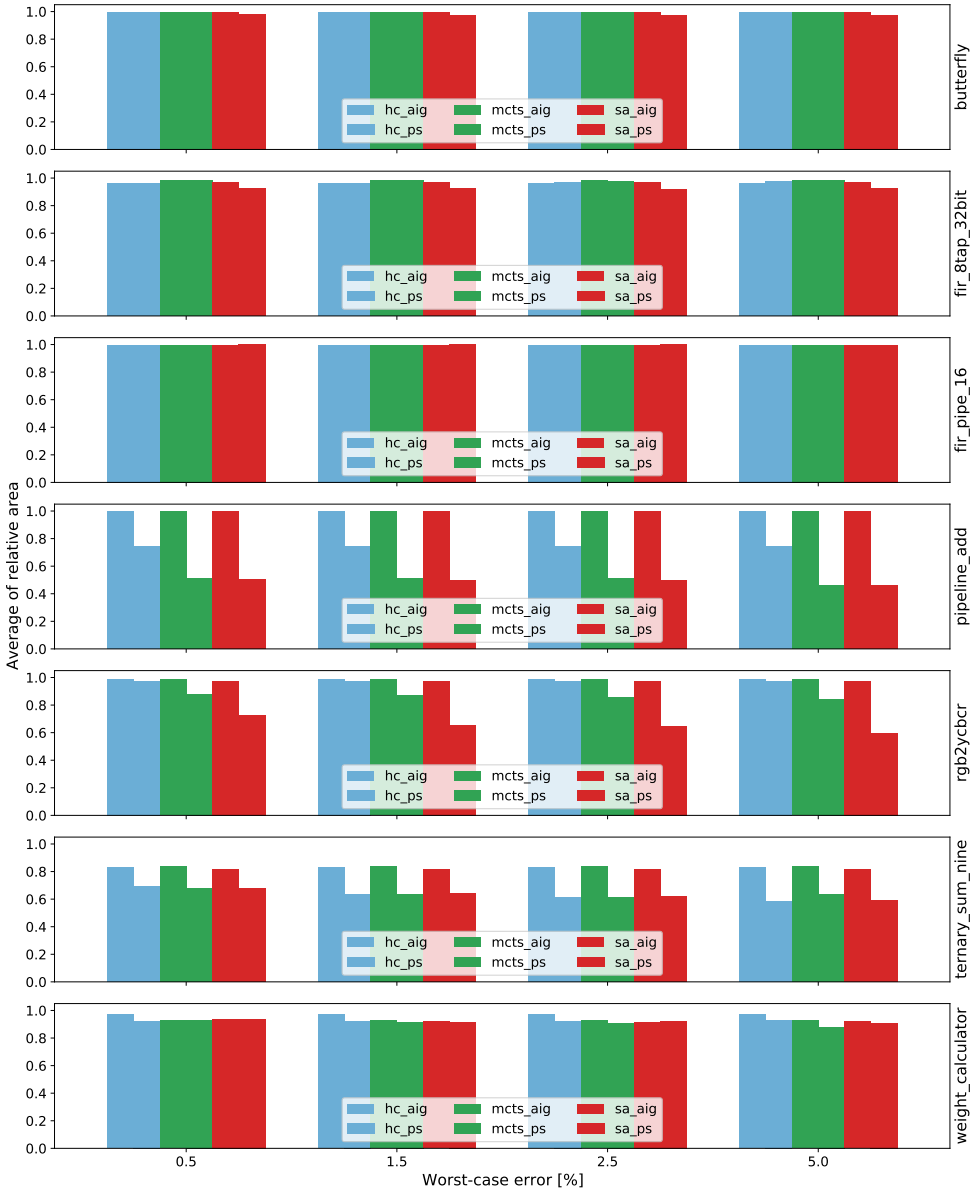


Figure 3.7: Resulting average of the relative area. Adapted from [101].

Figure 3.7 also allows to study the impact of the search method on the quality of the approximate outcome, and the figure reveals differences between

the different search methods. In general, SA performs very well and achieves higher savings than HC – especially for the benchmarks `pipeline_add` and `rgb2ycbcr`. SA is more explorative than HC, allowing SA to escape a local minimum and explore larger parts of the search space, whereas HC's greedy approach seems to hinder exploration and thus achieved savings. The performance of MCTS lies between HC and SA. For improved performance, MCTS needs to compute more iterations, which would allow MCTS to explore more search tree branches to find better solutions. With higher budgets, MCTS can also trade off the exploration of branches with low visit counts and the exploitation of nodes which proved to be more rewarding in previous iterations (cf. experimental results in [101]). Awais et al. [6–9] conducted further research on MCTS in ALS.

Overall, our experiments show that the quality of the result highly depends on the approximation technique as well as on the search method. This underlines the necessity of conducting extensive experiments with different approximation and search techniques, which CIRCA well supports. Furthermore, our results also point to the fact that the achievable area savings strongly depend on the input design. For example, the benchmarks `butterfly` or `fir_pipe_16` describe challenging approximation problems for which all search methods could achieve only marginal area savings.

Table 3.4 lists the average runtimes of the experiments. The formal verification used for quality assurance dominates the runtime of the ALS process, which ranges from a few seconds up to several days, depending on the number of verifications performed and depending on the complexity of the occurring verification problems. However, due to randomness in the taken path through the search space and in the applied approximations, the complexity of the occurring verification problems may differ. Thus, the runtime of the ALS may vary even though the same number of verifications has been performed for the same benchmark circuit, e.g., the `butterfly` benchmark (cf. HC with AIG rewriting).

Comparing the runtimes of the three search methods reveals that HC often provides significantly shorter runtimes than SA or MCTS; however, as Figure 3.7 shows, at the cost of lower area savings on average. Furthermore, HC performs, in general, significantly fewer iterations compared to SA and MCTS. There are two explanations for this: The first relates to how we count iterations. Basically, we increase the iteration count once a node gets selected. Compared to HC, SA tends to accept more nodes which naturally increases its iteration count. MCTS performs an iteration when performing back propagation. This leads to more iterations, although the runtime is not increased significantly. Second, HC might get stuck in a local minimum quickly. While HC then terminates the search, SA and MCTS continue with more iterations (cf. `rgb2ycbcr`).

Concluding our experimental results, we demonstrated that CIRCA is general and modular, allows for a swift configuration of an ALS process, and thus qualifies as an experimental environment for conducting research in the field of approximate computing. Furthermore, for the ALS process, we identi-

Table 3.4: CIRCA's average runtimes and number of selected nodes.

	Search method	Worst-case error bound [%] AIG rewriting 0.50		1.50		2.50		5.00		Precision scaling 0.50		1.50		2.50		5.00	
butterfly	HC	1:02:06	(8)	1:00:44	(8)	1:37:15	(8)	1:56:42	(8)	1:05:10	(9)	1:06:53	(9)	1:03:16	(9)	1:12:04	(9)
	MCTS	4:26:59	(94)	4:31:09	(94)	4:27:34	(94)	4:26:58	(94)	3:50:56	(95)	3:48:28	(95)	3:50:12	(95)	3:54:07	(95)
	SA	13:32:12	(180)	13:33:15	(180)	13:45:41	(180)	15:15:19	(180)	11:12:22	(114)	9:02:28	(114)	7:24:04	(116)	7:30:44	(122)
fir_8tap_32bit	HC	4:22:32	(22)	4:19:26	(22)	5:26:55	(22)	4:21:40	(22)	11:11:03:29	(59)	12:16:51:15	(64)	8:10:27:19	(41)	3:14:55:14	(25)
	MCTS	4:55:51	(100)	4:52:37	(100)	4:52:19	(100)	5:01:21	(100)	9:14:51:55	(100)	7:23:22:51	(100)	8:11:59:16	(100)	2:00:51:45	(100)
	SA	11:37:48	(80)	13:56:32	(80)	11:47:47	(80)	13:51:39	(80)	3:14:04:07	(80)	3:12:22:52	(80)	2:04:25:36	(80)	2:07:08:38	(80)
fir_pipe_16	HC	2:44:43	(7)	2:51:59	(7)	2:42:33	(7)	1:14:19	(7)	4:41:54	(66)	4:09:07	(66)	4:10:31	(66)	5:00:36	(66)
	MCTS	7:03:34	(100)	8:20:24	(100)	7:50:02	(100)	7:52:15	(100)	3:41:17	(100)	3:35:20	(100)	3:33:54	(100)	3:42:44	(100)
	SA	1:01:12:54	(154)	22:49:41	(154)	20:43:53	(154)	1:00:43:35	(154)	8:15:25	(108)	12:16:37	(118)	10:01:33	(111)	10:17:55	(111)
pipeline_add	HC	1:04	(1)	1:03	(1)	2:42	(1)	2:40	(1)	6:32	(16)	6:37	(16)	6:39	(16)	9:36	(16)
	MCTS	2:35	(41)	2:34	(41)	2:34	(41)	3:01	(41)	6:19	(92)	6:32	(94)	5:58	(89)	5:36	(83)
	SA	5:03:45	(540)	6:45:50	(540)	4:53:54	(540)	3:18:25	(540)	33:31	(40)	15:29	(40)	19:16	(38)	22:04	(41)
rgb2ycbcr	HC	16:09	(3)	16:32	(3)	11:08	(3)	10:48	(3)	22:07	(11)	1:50:57	(11)	1:48:45	(11)	55:15	(11)
	MCTS	2:13:00	(97)	1:37:42	(97)	1:44:30	(97)	1:28:23	(97)	3:05:08	(100)	3:52:15	(100)	4:42:26	(100)	10:20:47	(100)
	SA	2:57:32	(118)	3:59:37	(118)	6:27:43	(118)	10:06:10	(118)	1:09:26	(84)	2:41:00	(93)	1:38:51	(94)	4:24:12	(99)
ternary_sum_nine	HC	54:35	(22)	45:54	(22)	36:05	(22)	42:17	(22)	1:48:21	(80)	1:08:33	(99)	1:08:27	(101)	1:34:12	(140)
	MCTS	1:24:25	(99)	51:28	(99)	41:36	(99)	27:40	(99)	10:38	(100)	10:33	(100)	10:34	(100)	10:50	(100)
	SA	2:23:13	(100)	4:12:25	(100)	3:28:07	(100)	3:20:30	(100)	19:52	(51)	38:23	(58)	37:28	(62)	47:41	(63)
weight_calculator	HC	13:02	(2)	12:23	(2)	12:05	(2)	5:58	(2)	1:12:23:30	(11)	11:42:54	(8)	23:14:32	(9)	22:00	(3)
	MCTS	5:10:43:45	(65)	3:10:49:07	(99)	2:22:10:17	(100)	14:54:21	(98)	2:01:19:13	(33)	4:21:43:29	(84)	4:20:36:04	(99)	4:13:20:58	(99)
	SA	1:22:11:05	(17)	2:13:52:05	(33)	1:14:49:32	(34)	1:06:34:56	(33)	22:51:42	(9)	1:18:00:59	(16)	1:09:53:11	(17)	1:06:54:26	(19)

Runtimes are shown in the format days:hours:minutes:seconds and are followed by the average number of performed iterations.

fied that 1) the search method as well as 2) the approximation technique affect the outcome of the ALS, and 3) formal verification significantly contributes to the ALS runtime. Consequently, each of the main steps of the general ALS process described in Section 2.1 impacts the achieved improvements in the target metric and/or the runtime of the ALS process; thus, the experiments motivate the different research directions of this dissertation.

The approach in Chapter 4, jump search, considers the findings 1) and 3) to enhance the overall ALS efficiency by considering domain knowledge in the search. Chapter 5 devotes itself to finding 2) and proposes MUSCAT, a novel approximation technique. Chapters 6 and 7 consider finding 3). The concept of proof-carrying approximate circuits in Chapter 6 seeks to render a full verification process for library components obsolete by exploiting the concept of proof-carrying hardware. Chapter 7 presents a pre-processing step that pre-verifies the search space for the ALS process to omit verifications during ALS.

3.6 CONCLUSION

In this chapter, we have presented the flexible open-source ALS framework CIRCA. We have elaborated on CIRCA's architecture that incorporates the key requirements that we identified from related work: general, modular, compatible, extensible, and open-source. Through its architecture, CIRCA qualifies as a research tool for approximate computing to conduct comparative studies of different methods and techniques under static conditions.

Our experimental results have demonstrated that CIRCA allows for swift changes in the ALS setup, confirming that CIRCA indeed satisfies the key requirements. Furthermore, we formulated three main findings that motivate the different research directions presented in Chapters 4 to 7 of this dissertation.

4

JUMP SEARCH: FAST SYNTHESIS OF APPROXIMATE CIRCUITS

4.1 OVERVIEW

In the last years, several automated approximate logic synthesis (ALS) methodologies [10, 50, 65, 69, 91, 94] have been presented that state the synthesis process as an optimization problem with the target metrics as objectives and a user-defined error bound as a constraint (see Section 2.1). A major issue for this search or optimization problem is that only very few general assumptions can be made about the design space except that typically it is huge, rendering exact optimization techniques infeasible. Thus, to find suitable solutions, the methodologies employ search-based methods that iteratively expand the design space step-by-step. The found solutions are evaluated and ranked with respect to their error or output quality. Due to the vast design space, a large number of solutions should be evaluated to increase the chance of finding a suitable approximate circuit (AxC). Quality assurance is either done via formal verification or testing, and both require considerable runtime, making the quality assurance step often a bottleneck in automated approximate logic synthesis. Additionally, many of the evaluated solutions are not of interest since their quality is unacceptable.

The related work discussed in Section 2.1 and the approaches in CIRCA (see Section 3.5) heavily rely on circuit parameters, such as hardware area, delay, power consumption, or quality. The parameters and the quality are determined via synthesis and verification and then usually incorporated into heuristic functions either to guide the search, e.g., ASLAN [69], or to evaluate randomly generated AxCs, e.g., SCALS [50], leading to considerable runtimes of the approximation process. In fact, the employed search methods in current ALS processes spend their search budget uniformly throughout the search, i.e., the search does not utilize domain knowledge, which causes the search to spend the search budget on parts of the design space that are not of interest. Thus, we present the jump search (JS) methodology in this chapter, which performs a guided, non-uniform sub-sampling of the design space, and takes domain knowledge into account to spend the search budget on the most promising parts of the design space.

We present jump search as an approach that minimizes the required number of evaluations, i.e., syntheses and verifications, and thus, enables a rapid ALS process. Instead of iteratively expanding the search space and

evaluating solutions step-by-step, jump search uses a different scheme and proceeds in three phases:

1. *Pre-processing*: We determine so-called *impact factors* for all suitable approximation candidates in the input design. The impact factors characterize a candidate's effect on the target metric – hardware area in our case – and on the error at the output of the circuit.
2. *Path planning*: We generate a path in the search space by iteratively applying a heuristic function that selects the next solution based on the pre-computed impact factors. While the solutions along the path are more and more aggressively approximated, they are not evaluated for their quality or target metric.
3. *Binary search*: We perform a binary search on the path and now evaluate solutions through synthesis and verification to find the one AxC that applies approximations most aggressively while still satisfying the user-defined quality constraint.

Key to jump search is that costly evaluations, i.e., syntheses and verifications, are performed only in the third phase and due to the binary search technique, which allows us to literally *jump over* solutions on the path, on a relatively small number of solutions compared to other heuristic search methods, resulting in a significantly reduced runtime of the approximation process. We follow the commonly made assumption [22, 41, 69, 85] that more aggressive approximations lead to more significant improvements in the target metric, which might not always hold. Thus, jump search might actually miss the best solution on the path and, since jump search bases on heuristics, there is obviously no guarantee that we find the optimal solution in the overall design space. However, jump search achieves similar improvements in hardware area as the commonly used search techniques simulated annealing or hill climbing, but at a greatly reduced runtime.

We have presented jump search first at a workshop [108] and then in a conference publication [102]. This dissertation extends the previously presented jump search, and studies different and more sophisticated techniques for calculating impact factors and path generation. We present the approach and assess several techniques for realizing the pre-processing and path planning phases. In the pre-processing phase, we study more straightforward constant area impact factors and more involved area impact factors that functionally depend on a candidate's worst-case (WC) error. For calculating suitable error impact factors, we experiment with different feature ranking methods, namely least absolute shrinkage and selection operator (LASSO), Hilbert-Schmidt independence criterion LASSO (HSIC LASSO), and random forest (RF). In the path planning phase, we employ two different heuristics that differ in the way area reduction and error increase are considered when selecting the next node in the path. In summary, we make the following contributions with jump search:

- We present jump search as a methodology for rapid approximate logic synthesis, operating in three phases.

- We propose to exploit candidate information on the area and error to select a subset of promising approximate circuits from the search space.
- We demonstrate in our experimental results that jump search remains competitive to the commonly used search methods simulated annealing and hill climbing while enduring significantly shorter runtimes; in particular, jump search achieves speed-ups of up to 468× in our experiments.

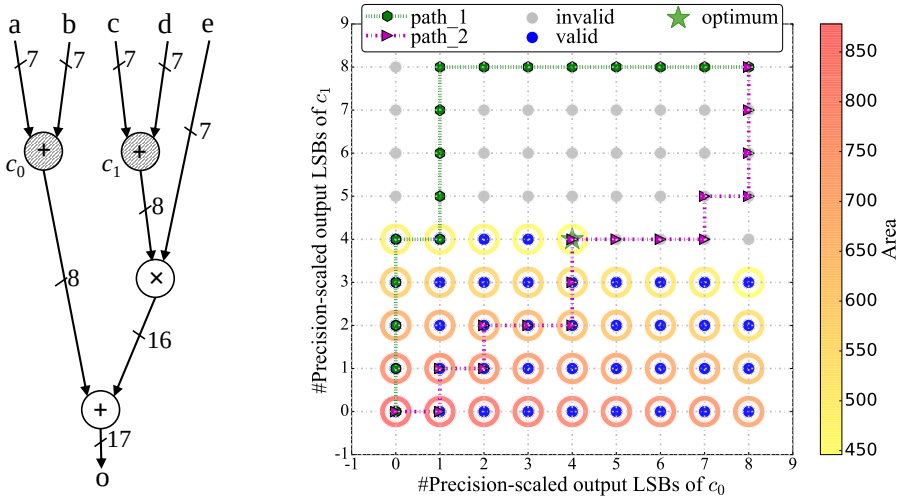
The remainder of this chapter is structured as follows: In Section 4.2, we first provide a motivational example to derive the concept of jump search. Then, we present in Section 4.3 the algorithm in pseudo-code form to explain jump search's three phases in detail: pre-processing, path planning, and binary search. Section 4.5 presents an approach to improve the accuracy of the estimation of the impact factor of the area. Section 4.6, contributed by my colleague, Hassan Ghasemzadeh Mohammadi, discusses the feature ranking methods used to determine the impact factor for the candidate's error. The figure-of-merits (FoMs) and the integration of the impact factors into the FoMs are discussed in Section 4.7. Section 4.8 details the implementation of jump search's tool flow. The experimental results are presented in Section 4.9 and detail different aspects of jump search. Finally, Section 4.10 concludes the chapter.

Multiple people have been involved in the development and implementation of jump search. My colleague, Hassan Ghasemzadeh Mohammadi, provided the idea of utilizing feature ranking methods to determine a candidate's impacts on the overall circuit error and contributed Section 4.6. My student research assistant, Matthias Artmann, contributed to the implementation, and Khushboo Chandrakar implemented different feature ranking methods and provided a comparison of the methods in her Master's thesis [21]. My contribution to this work lies in developing the overall concept and methodology, and in the implementation.

4.2 MOTIVATIONAL EXAMPLE AND CONCEPTUAL OVERVIEW

Figure 4.1a shows the data flow graph of an exemplary circuit that implements a data path with four arithmetic components. The two adders, c_0 and c_1, have been selected as candidates for approximation. Precision scaling is employed as approximation technique and a normalized WC error bound of 1.5% is imposed as the quality constraint upon the output o. Figure 4.1b shows the resulting decision space or search space, respectively, spanned by applying precision scaling to both candidates from zero bits, which denotes the unapproximated adder, to eight bits, which means the adder is completely removed. We explored the search space exhaustively, and determined the overall area and the WC error for each design point. We have used the number of four input lookup tables (4-LUT) after circuit synthesis with ABC [14] as area metric and evaluated the WC error by formal verification. In Figure 4.1b, blue dots represent AxCs that adhere to the quality constraint,

and gray dots represent the invalid ones that show a limit-exceeding WC error; colors indicate the area (cf. heat map).



(a) Example circuit.

(b) Visualization of the search space and paths.

Figure 4.1: Examplary circuit (a) along with its search space (b). The approximation candidates of the circuit are highlighted in gray. To span the design space, precision scaling is utilized as approximation method and a WC error of 1.5% of the maximal output value is used as quality constraint. Out of the 81 circuits that form the design space 42 are valid.

Intuitively, looking at the circuit in Figure 4.1a, one would expect that approximating candidate c_1 leads to higher area savings than approximating candidate c_0 since synthesis tools will reduce both the subsequent multiplier and the adder after precision-scaling candidate c_1. On the other hand, due to the data flow, the impact of candidate c_1's inaccuracy on the WC error of the output o is higher than the impact of candidate c_0. Figure 4.1b underlines the intuition: While approximating all output bits of candidate c_0 still leads to a valid approximate circuit, we must keep at least the four most significant bits of candidate c_1's output to satisfy the quality constraint. The optimal solution (green star), i.e., the valid AxC with the smallest area, is found in this case when balancing the approximations of the candidates.

For more complex circuits, the search space grows extremely large, rendering an exhaustive search infeasible. Moreover, a substantial part of the search space can represent design points that do not meet the quality constraint. Hence, many related frameworks [22, 69, 94, 101] proposed iterative techniques that expand the search space step-by-step, starting from the unapproximated circuit. There are two issues with such techniques: First, for each new AxC, synthesis and quality assurance need to be run, which are very time-consuming. Second, it is not apparent how the search space should be expanded to find a suitable solution within a given time budget.

Jump search considers the described intuition from the example to achieve low runtimes while still finding suitable AxCs that meet the quality constraint. Central to JS is the concept of a *path* in the search space. A path always starts at

the original, unapproximated circuit and ends at an AxC with all candidates fully approximated. For example, in Figure 4.1, the path's starting point is at $(0,0)$ and the path's endpoint at $(8,8)$, when both candidates c_0 and c_1 have been removed. The figure displays two possible paths through the search space, $\mathtt{path}_1$ and $\mathtt{path}_2$. Walking along path $\mathtt{path}_1$ means to apply first a sequence of approximations to candidate c_1, which quickly reduces area but also quickly leads to invalid design points. In contrast, path $\mathtt{path}_2$ balances the importance of both the candidates' impact on area and quality.

JS creates a path in the search space by employing a heuristic, called *figure-of-merit*, incorporating the impact of the candidates on the target metric and quality through *impact factors*. The impact factors are candidate-specific and determined once in a pre-processing phase. Importantly, JS determines the path without referring to expensive synthesis and quality assurance, solely relying on the impact factors.

After creating the path, JS searches for the best AxC on the path, which is the one that minimizes the target metric and satisfies the quality constraint. As the degree of approximation increases along the path, we assume that AxCs deeper on the path generally achieve higher savings. To that end, JS performs a binary search to efficiently find the deepest (or last) valid AxC on the path. Costly synthesis and quality assurance steps are only performed for the inspected nodes on the path.

4.3 JUMP SEARCH METHODOLOGY

This section gives an overview of the jump search technique and its three phases: *pre-processing*, *path planning*, and *binary search*. We focus on a JS implementation (see Section 4.8) that uses hardware area as target metric, employs precision scaling as approximation technique, relies on the worst-case error as quality metric, and utilizes formal verification for quality assurance.

Algorithm 4.1 shows the pseudo-code of JS. The algorithm takes as inputs the original circuit $\mathtt{i}$, the set of candidates for approximation C, and the user-defined quality constraint T_{WC}, which we consider to be a worst-case error threshold. JS assumes that either the user or automated methods (see. Section 2.1) provide the set of candidates C. Then, JS proceeds in the phases described in more detail in the following sections.

4.3.1 *Pre-processing Phase*

Jump search's pre-processing phase, shown in Lines 2 to 4 of Algorithm 4.1, determines the impact factors for the candidates $c \in C$ concerning area and error, $\mathtt{if}_{\mathtt{area}}(c)$ and $\mathtt{if}_{\mathtt{err}}(c)$. Different methods and approaches for defining and computing these impact factors are discussed in detail in Sections 4.5 and 4.6.

Algorithm 4.1: Pseudo-code of the jump search algorithm.

Input: Original circuit i, set of candidates C, global WC error threshold T_{WC}
Output: Approximate circuit AxC

```
1  Function jumpSearch( i, C, T_WC ):
       /* Pre-processing phase                                          */
2      foreach c ∈ C do
3          if_area(c) ← computeImpactOnArea( c, i )
4          if_err(c) ← computeImpactOnError( c, i )
       /* Path planning phase                                           */
5      P ← ∅                                  // Path, each node represents an AxC
6      s ← i                                        // Start from original circuit
7      while ∅ ≠ S ← determineDirectSuccessorNodes( s ) do
8          bestFoM ← −inf
9          foreach s_s ∈ S do
10             f ← FoM( s, s_s, C )              // evaluate node with heuristic FoM
11             if bestFoM < f then
12                 bestFoM ← f
13                 s ← s_s
14         P ← P.append( s )                          // Add best node to path
       /* Binary search phase                                           */
15     low ← 0
16     high ← length(P) − 1
17     while low ≠ high do
18         mid ← ⌈(low+high)/2⌉
19         AxC.area ← synthesize( P[mid] )          // Determine target metric
20         AxC.valid ← verify( i, P[mid], T_WC )              // Verify quality
21         if AxC.valid then
22             low ← mid
23         else
24             high ← mid − 1
25     return AxC
```

4.3.2 *Path Planning Phase*

Jump search's path planning phase, shown in Lines 5 to 14 of Algorithm 4.1, creates a path through the search space, starting from the original circuit and ending at the fully approximated circuit. In each iteration, jump search determines all direct successor nodes that represent AxCs with a lowered local quality constraint (LQC) $\epsilon(c)$ of a candidate c by one step (see Section 3.4.2). Thus, the number of candidates defines the dimensionality of the search space.

The concept of local quality constraints enables jump search to expand the search space in a controlled way and uniquely define AxCs. Figure 4.2 visualizes JS for a scaled-down version of the circuit in Figure 4.1a, where the two candidates, c_0 and c_1, have only four-bit outputs so that the search space comprises a total of 25 nodes. Path planning starts from the error-free circuit, i.e., the candidates satisfy the LQCs $\epsilon(c_0) \leqslant 0$ and $\epsilon(c_1) \leqslant 0$. Direct successor nodes are the two AxCs with the LQCs $\{\epsilon(c_0) \leqslant 1, \epsilon(c_1) \leqslant 0\}$ and

$\{\epsilon(c_0) \leqslant 0, \epsilon(c_1) \leqslant 1\}$. Which AxC is chosen as the best successor is determined by the heuristic FoM (see Section 4.7) that takes the pre-computed impact factors into account. As the FoM only depends on the pre-computed impact factors, jump search omits synthesis or verification steps completely and plans the path quickly. Path planning completes when no more successors are available, i.e., all candidates are fully approximated. The blue nodes in the center of Figure 4.2 highlight a fully planned path and, in the example, the fully approximated circuit relates to the LQCs $\{\epsilon(c_0) \leqslant 15, \epsilon(c_1) \leqslant 15\}$.

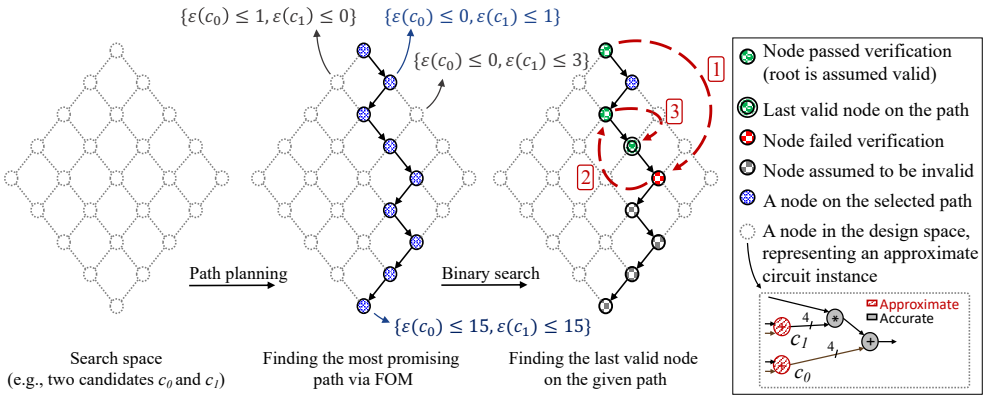


Figure 4.2: Visualization of the jump search phases. The circuit shown is a scaled-down version of the example of Figure 4.1a, where the two candidates c_0 and c_1 have four-bit outputs. Note that due to the scaled-down bit widths, jump search generates a path different from the one shown in Figure 4.1b.

4.3.3 *Binary Search Phase*

In the last phase, jump search performs a binary search on path P to find the deepest (or last) AxC that still satisfies the user-defined WC error threshold T_{WC}, shown in Lines 15 to 24 of Algorithm 4.1; only in this phase, JS finally employs synthesis and verification steps for evaluating the AxCs.

The right-hand side of Figure 4.2 visualizes the binary search phase. Path P is highlighted in blue, the valid AxCs are highlighted in green, and red highlights AxCs deemed invalid through verification. Binary search first visits the fifth node on path P, which represents an AxC where one bit of candidate c_0 and three bits of candidate c_1 have been precision-scaled. The AxC does not pass the quality assurance and is thus invalid. As a result, jump search assumes all succeeding AxCs are invalid (checked gray nodes in the figure) since these AxCs exhibit an even higher degree of approximation, and thus, presumably even larger errors. Next, binary search visits the third node on path P, which is valid, and finally, the fourth node, which is also valid and returned by JS as the deepest valid AxC.

4.4 SEARCH TECHNIQUES AND THEIR LIMITATIONS

This section discusses the size of the search space, the differences between search techniques, and their potential limitations. Assuming a circuit with n candidates, where each candidate $c_i, i \in [0, n-1]$ has b_i output bits that can be precision-scaled, the total number of nodes in the search space n_{AxC} is given by:

$$n_{AxC} = \prod_{i=0}^{n-1} (b_i + 1) \tag{4.1}$$

The search space is exponential in the number of candidates and forms a lattice with the number of candidates n defining the number of dimensions, each extending to the range $[0, b_i]$ (cf. Section 3.4.2). The longest path in the lattice visits

$$1 + \sum_{i=0}^{n-1} (b_i) \tag{4.2}$$

nodes.

Evaluating a node in the search space requires generating the circuit in a synthesis step to determine the area, followed by a verification step to verify the circuit's quality. Since these runs are computationally expensive – especially formal verification – exhaustive search becomes impractical for larger circuits, and randomly selecting nodes in the search space is presumably not effective with a limited compute time budget.

Since the search space expands towards increasing local WC errors of the candidates, starting from the unapproximated circuit, we actually deal with a search problem in a directed acyclic graph. For this kind of problem, related work applied search techniques such as hill climbing and simulated annealing [69, 94, 101].

Hill climbing as a greedy local search technique evaluates all direct successor nodes of a given node and moves on to the valid successor that minimizes the circuit's target metric but can easily get stuck when all successor nodes are invalid or no valid successor node improves the target metric. While the number of required synthesis and verification steps for hill climbing strongly depends on how long the search continues, a conservative upper bound can be given as follows: Along the longest path in the lattice, each expansion step evaluates n successor nodes except for the last expansion step where only the fully approximated node is left. This amounts to

$$1 + \underbrace{\sum_{i=0}^{n-1} (b_i)}_{\text{AxCs on path}} + \underbrace{(n-1)\left(\sum_{i=0}^{n-1} (b_i) - 1\right)}_{\text{Additional evaluations at each step, excluding the last}} \tag{4.3}$$

synthesis and verification steps, considering that we also need one synthesis (and verification) run to determine the area (and quality) of the unapproximated circuit.

Simulated annealing randomly selects one successor node. If the node is valid and improves the area, it is selected; if it does not improve the area, it is selected with a certain probability or rejected otherwise. If the node is not accepted, the next successor node is randomly selected and evaluated. Eventually, if no successor node is accepted, the search terminates. As a result, simulated annealing can escape local minima and typically requires fewer synthesis steps than hill climbing. However, in the worst case, simulated annealing always accepts the last evaluated node in each expansion step, giving simulated annealing the same upper bound for synthesis and verification steps as hill climbing (HC).

Jump search differs from hill climbing and simulated annealing in that it does not run synthesis and verification during the path planning phase but applies fast heuristics to select the next node and create the longest path in the search space. In the subsequent binary search phase, jump search inherits the complexity of the binary search and evaluates

$$\left\lceil \log_2 \left(1 + \sum_{i=0}^{n-1} b_i\right) \right\rceil \tag{4.4}$$

synthesized and verified nodes. In addition, jump search performs synthesis in its pre-processing phase to determine the area impact factors. Depending on the actual method chosen, this ranges from $n+1$ to $1+\sum_{i=0}^{n-1}(b_i)$ synthesis steps, including the initial synthesis of the unapproximated circuit. In our experiments, we have limited this effort to $4n+1$ runs (see Section 4.9). The determination of the error impact factors does not require additional synthesis. Instead, the circuit is simulated functionally for a set of input vectors, and a feature ranking method is applied. Importantly, neither the computation of $\mathtt{if}_{area}$ nor $\mathtt{if}_{err}$ requires formal verification, and our experimental results show that the runtime for pre-processing is low compared to the overall runtime (see Section 4.9.4).

The difference in synthesis and verification steps between hill climbing and simulated annealing, on the one hand, and jump search, on the other hand, is pronounced. For example, assuming a circuit with 20 candidates with 32-bit outputs, hill climbing and simulated annealing would require 12 782 synthesis and verification steps in the worst case, whereas jump search requires only ten synthesis and verification steps plus 128 synthesis-only steps for pre-processing.

The jump search technique proposed in this chapter, as well as hill climbing and simulated annealing, also have limitations as they assume that both the global error and the overall area savings are monotonically increasing when going down to deeper nodes in the search space. Since we use precision scaling as approximation technique and the WC error as error metric, this assumption certainly holds for the global error. However, other approximation techniques and error metrics might violate the assumption about

error monotonicity, and then jump search and related techniques need to be revisited and adapted. In addition, the area savings are not strictly increasing since circuit synthesis tools, including technology-mapping to 4-LUTs, can sometimes better exploit optimization potential for larger circuits than for smaller circuits. In our experience, this effect is not pronounced except for very small circuits (cf. Section 4.5 and Figure 4.3).

4.5 ESTIMATING A CANDIDATE'S IMPACT ON AREA

This section presents the computation of the area impact factor $\mathrm{if}_{\mathrm{area}}(c)$, which expresses the impact on the overall circuit area when a candidate c is approximated. To achieve a metric with an intuitive interpretation and comparability between different candidates and approximation degrees, we define the area impact factor as area savings achieved by a specific approximation of a particular candidate relative to the area of the original circuit. In this way, $\mathrm{if}_{\mathrm{area}}(c)$ lies in the range $[0, 1]$, where the lower bound of zero models the area impact without any approximation and the upper bound of one indicates approximations that obliterate the circuit. $\mathrm{if}_{\mathrm{area}}(c)$ is obviously dependent on the original circuit and the specific candidate that undergoes approximation, but also on the applied approximation technique and the degree of approximation that the candidate's local quality constraint controls.

The function *computeImpactOnArea*(c, i) in Line 3 of Algorithm 4.1 involves a trade-off between accuracy and computational effort. The most accurate characterization of a candidate's impact on the overall area is achieved by computing it for all possible approximation degrees, i.e., for all possible bit truncations when precision scaling is used. Since these approximation degrees correspond to different LQCs $\epsilon(c)$, the area impact factor turns into the function $\mathrm{if}_{\mathrm{area}}(c, \epsilon(c))$. In order to determine this function, jump search's pre-processing phase needs to perform a synthesis run for each candidate and approximation degree. In many cases, this will be prohibitively expensive and contradict jump search's design goal of a rapid synthesis process.

Figure 4.3 shows an example for the benchmark `ternary_sum_nine`, where precision scaling approximates the candidate c, an adder from the data path with 18 output bits. The gray dots visualize all data points that can be acquired for the candidate, i.e., each data point represents the candidate with a different number of precision-scaled output bits; note that the area savings are not monotonically increasing, as pointed out in Section 4.4. The green and the blue curve represent two possible functions of $\mathrm{if}_{\mathrm{area}}(c, \epsilon(c))$, both determined via curve fitting but with different samples.

Jump search reduces the required synthesis steps during pre-processing by inspecting the area savings only at a subset of approximation degrees and applying curve fitting to derive an estimation function for $\mathrm{if}_{\mathrm{area}}(c, \epsilon(c))$. For the benchmarks in our experiments, which provide adders and multipliers as candidates and precision scaling as approximation technique, we found that the power function $a(\epsilon(c))^b$ mostly provides a good model for $\mathrm{if}_{\mathrm{area}}(c, \epsilon(c))$. Naturally, the number and position of data points used for fitting significantly

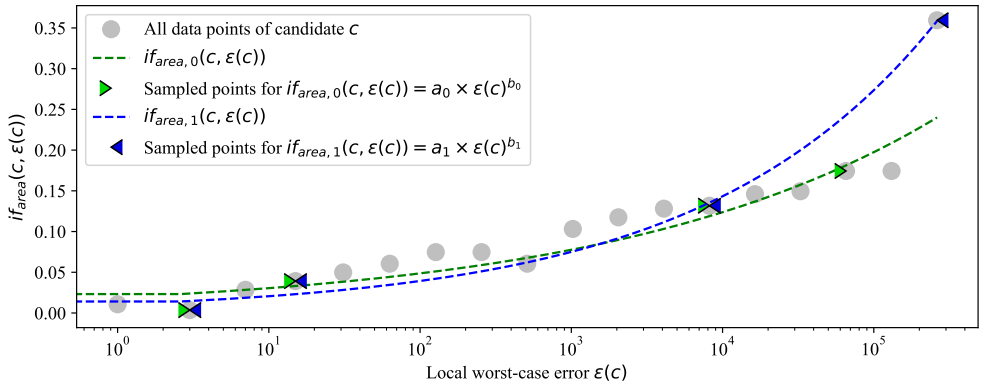


Figure 4.3: Relative area savings achieved for the benchmark `ternary_sum_nine` through precision-scaling an adder with an 18 bit output. The gray dots show all data points that can be acquired with precision scaling for the candidate. The green and blue triangles are the data points which have been used to determine the green fitting curve ($a_0 = 0.0191, b_0 = 0.203$) and the blue fitting curve ($a_1 = 0.0108, b_1 = 0.281$), respectively.

impact the fitting function. The dashed green and blue lines in Figure 4.3 present the fitting functions determined with five data points, comprising the area impact factor of the original circuit and four approximated versions of the candidate c. While the data points for generating the blue line include a fully approximated candidate c with $\epsilon(c) = 2^{18} - 1$, the green line was generated with less aggressive approximations. As a result, the blue line over-estimates the achievable area savings for more aggressive approximations and is less accurate than the green line for small and moderate approximations. Since aggressive approximations, such as a fully approximated candidate, are unlikely to occur in real applications, we argue that the accuracy for less and moderate approximations should be emphasized. Hence, in our experiments with benchmark circuits, we have omitted the data point corresponding to full approximation and selected data points where 0%, $\approx$ 12.5%, $\approx$ 25%, $\approx$ 70%, and $\approx$ 90% of the output bits are precision-scaled for curve fitting.

Another extreme approach for estimating $\mathrm{if}_{\mathrm{area}}(c)$ is to determine a single data point only, turning the area impact for a candidate into a constant. While this requires only one synthesis step per candidate, the resulting estimate will be somewhat inaccurate for certain approximations. In our experiments, we chose the data point representing the full approximation or maximal possible area savings, respectively. That is, $\mathrm{if}_{\mathrm{area}}(c)$ denotes the area of the overall circuit with candidate c being fully approximated. Even though this approach may lead to over-estimated area impacts, we apply this simple scheme since it over-estimates the area savings for all candidates, and it is even independent of the approximation method.

4.6 ESTIMATING A CANDIDATE'S IMPACT ON ERROR

In this section, we discuss the error impact factor $\mathrm{if}_{\mathrm{err}}(c)$, which expresses the impact on the overall circuit's error when a candidate c is approximated.

Our estimation method for $\mathrm{if}_{\mathrm{err}}(c)$ exploits feature ranking methods and is exemplified in Figure 4.4. We apply d vectors of randomly generated data to the primary inputs of the circuit and sample the resulting values at the outputs of the n candidates and the primary output(s). The sampled values of the candidates, i.e., $\mathbf{X} \in \mathbb{R}^{d \times n}$, and values at the primary outputs, i.e., $\mathbf{y} \in \mathbb{R}^{d}$ for a single primary output, form a labeled dataset that is used for the training of the feature ranking methods. After the training, the feature ranking methods can rank the candidates based on their contribution to the error observed at the primary output(s) of the design. Since the corresponding correlations are in the range of $[-1, +1]$, we take their absolute values and create a list of candidates with associated $\mathrm{if}_{\mathrm{err}}(c) \in [0, +1]$, that is sorted according to non-decreasing values.

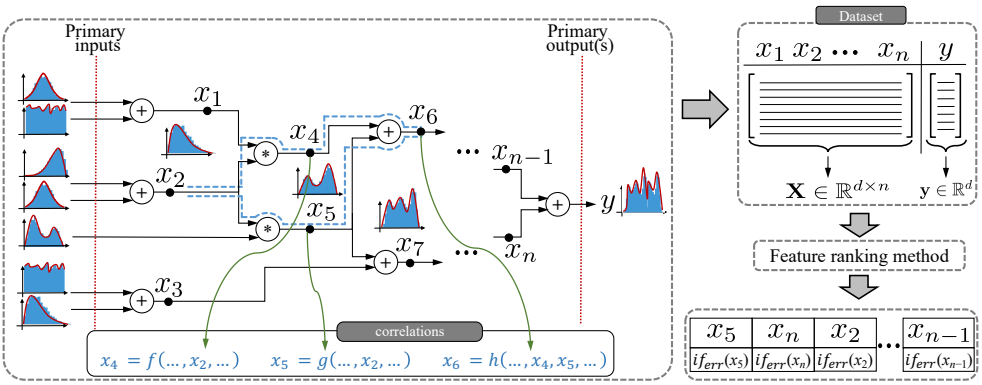


Figure 4.4: Training of feature ranking methods for the estimation of $\mathrm{if}_{\mathrm{err}}(c)$. The result is a list of candidates sorted according to non-decreasing contributions to the overall circuit's error, and corresponding error impact values $\mathrm{if}_{\mathrm{err}}(c) \in [0, +1]$.

A feature ranking method for estimating $\mathrm{if}_{\mathrm{err}}(c)$ is required to satisfy four properties: First, the quality of the ranking method should be independent of the distribution of training data. In fact, the ranking method should not impose any restrictions on the statistical distributions of the sampled values in the obtained dataset. This is a key requirement since the actual distribution depends on the concrete workload, which may not be available during the AxC synthesis step. Second, it is necessary to consider both linear and non-linear correlations among the sampled outputs of the candidates, i.e., $\mathbf{X}$, as well as the non-linearity between these values and the sampled primary output(s), i.e., $\mathbf{y}$. Correlations exist due to dependencies in the data flow graph, as shown for x6, x5, x4, and x2 in Figure 4.4. Such correlations become non-linear when non-linear arithmetic operators appear in the data flow graph, e.g., an absolute operator. Moreover, the correlation of errors is in general non-linear. Third, the ranking method should be able to handle multi-output functions, which requires that the underlying feature ranking considers the contribution of each component on the error of all primary outputs jointly. Last, the training time of the ranking method should be negligible in comparison with the path planning and the binary search

phases of JS. In the following, we introduce three statistical feature ranking methods that use different approaches to solve the ranking problem.

4.6.1 *Least Absolute Shrinkage and Selection Operator*

A popular approach for feature selection is to use regularized statistical models in which features are ranked according to their contributions to the underlying model. The least absolute shrinkage and selection operator (LASSO) [87] is a linear regression technique that exploits l_1-norm regularization to both select features and boost the regressor's accuracy. In our work, we have a labeled training dataset in which each pair, $(\mathbf{x}_i, y_i) \in \mathbb{R}^n \times \mathbb{R}$, has been generated independently of an unknown distribution. The goal of LASSO is then to find a regression of the feature vector, i.e., vector of candidates, to rank the labels $R : \mathbb{R}^n \longmapsto \mathbb{R}$, utilizing the most influential features. LASSO selects the features subject to the following objective function:

$$\underset{\mathbf{w}}{\operatorname{argmin}} \left(\frac{1}{2} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_1 \right) \tag{4.5}$$

where $\mathbf{y} \in \mathbb{R}^d$ is a sample vector of the primary output, $\mathbf{X} \in \mathbb{R}^{d \times n}$ is the sample matrix of n input features, i.e., candidate outputs, and $\mathbf{w} \in \mathbb{R}^n$ refers to the vector of regression coefficients. Moreover, $\lambda \in \mathbb{R}$ denotes the regularization hyper parameter, which is a non-negative real number that controls the values of the regression coefficients. In fact, λ controls the shrinkage of coefficients. A large value for λ decreases the number of non-zero elements in $\mathbf{w}$ and hence keeps the value of the objective function small. The linear regression along with the penalty term implies a consistent parameter ranking. In fact, the ranking of the features is preserved when λ increases. Thus, for our purposes, λ only needs to be tuned in a way that the number of non-zero coefficients is maximized. The obtained absolute values of the coefficients then represent the contribution of each feature to the output and are used as the feature selection metric.

The LASSO technique benefits a convex objective function with a linear constraint, which can then be efficiently solved. However, LASSO neglects non-linear dependencies among features, which is a limiting factor for our error impact factor estimation which involves non-linear correlations [59].

4.6.2 *Hilbert-Schmidt Independence Criterion LASSO*

HSIC LASSO [114] improves upon LASSO by replacing the mean square error part of the objective function with a non-linear error measure. This allows us to take into account the non-linear dependencies between features and the output variable:

$$\underset{\mathbf{w}}{\operatorname{argmin}}\Big(\frac{1}{2}\|\mathbf{R}_c - \sum_{i=1}^{n} w_i \mathbf{U}_c^{(i)}\|_F^2 + \lambda\|\mathbf{w}\|_1\Big) \quad \text{s.t. } \forall w_i \geqslant 0 \tag{4.6}$$

where $\|.\|_F$ is the Frobenius norm, $\mathbf{R}_c \in \mathbb{R}^{d\times d}$ and $\mathbf{U}_c^{(i)} \in \mathbb{R}^{d\times d}$ are cartelized Gram matrices of $\mathbf{R}_{j,k} = R(x_{i,j}, x_{i,k})$ and $\mathbf{U}_{j,k} = U(y_{i,j}, y_{i,k})$. The Gram matrices are obtained by:

$$\begin{aligned}\mathbf{R}_c &= (\mathbf{I}_d - \frac{1}{d}\mathbf{1}_d\mathbf{1}_d^\top)\mathbf{R}(\mathbf{I}_d - \frac{1}{d}\mathbf{1}_d\mathbf{1}_d^\top)\\ \mathbf{U}_c &= (\mathbf{I}_d - \frac{1}{d}\mathbf{1}_d\mathbf{1}_d^\top)\mathbf{U}(\mathbf{I}_d - \frac{1}{d}\mathbf{1}_d\mathbf{1}_d^\top)\end{aligned} \tag{4.7}$$

where $\mathbf{I}_d$ is the d-dimensional identity matrix and $\mathbf{1}_d$ is the d-dimensional vector of all ones. $R(y, y')$ and $U(x, x')$ are kernel functions which provide non-linearity to the model. They operate in a higher-dimensional space without computing the coordinates of the data in that space, but by solely computing the inner products between the images of all pairs of data in that space. The common choice for these kernel functions is the Gaussian kernel, in which the similarity of a pair of input samples, e.g., (x, x'), is computed via:

$$K(x, x') = \Big(-\frac{(x - x')^2}{2\sigma_x^2}\Big) \tag{4.8}$$

where σ_x is a hyper parameter and denotes the width of the Gaussian kernel.

The first term in Equation (4.6) represents the distance between the Gram matrices of features and the output variable. The second term, i.e., the penalty term, penalizes those features whose Gram matrices do not contribute to the estimation of the output Gram matrix. The first term can be written as:

$$\begin{aligned}\frac{1}{2}\|\mathbf{R}_c - \sum_{i=1}^{n} w_i \mathbf{U}_c^{(i)}\|_F^2 &= \frac{1}{2}\mathrm{HSIC}(\mathbf{y}, \mathbf{y})\\ -\sum_{i=1}^{n} w_i \mathrm{HSIC}(x_{*,i}, \mathbf{y}) &+ \frac{1}{2}\sum_{i,j=1}^{n} w_i w_j \mathrm{HSIC}(x_{*,i}, x_{*,j})\end{aligned} \tag{4.9}$$

where $x_{*,i}$ denotes the i-th column of the feature matrix, and the HSIC operator refers to the *Hilbert-Schmidt independence criterion* that measures the independence of two random variables as follows:

$$\mathrm{tr}(\mathbf{U}_c^{(i)}\mathbf{R}_c) \tag{4.10}$$

where $\mathrm{tr}(.)$ stands for the trace.

HSIC always gives a positive value for two random variables and its magnitude directly depends on the correlation between the variables. If two

random variables are statistically independent, HSIC returns 0. The first term in Equation (4.9), i.e., $\mathrm{HSIC}(\mathbf{y}, \mathbf{y})$, becomes constant and can be dropped from the optimization. The second term, i.e., $\mathrm{HSIC}(x_{*,i}, \mathbf{y})$, computes the correlation between the i-th feature and the output variable. If the i-th feature has a significant impact on the output, $\mathrm{HSIC}(x_{*,i}, \mathbf{y})$ takes a large value, and thus, w_i needs to take a large value to minimize the objective function in Equation (4.6). Otherwise, if the i-th feature has a negligible impact on the output, $\mathrm{HSIC}(x_{*,i}, \mathbf{y})$ leads to a small value close to zero, and its corresponding coefficient, i.e., w_i, shrinks to zero due to the l-norm. Thus, the important features that have a large impact on $\mathbf{y}$ are selected. In case of highly dependent features, e.g., x_i and x_j, the third term, i.e., $\mathrm{HSIC}(x_{*,i}, x_{*,j})$, results in a large value. Since the corresponding coefficients, i.e., w_i and w_j, are positive, they tend to be eliminated by the l-1 regularizer. This means that redundant features are not selected by HSIC LASSO.

Unlike LASSO, HSIC LASSO takes into account both linear and non-linear dependencies, not only among input variables but also between the input and output variables.

4.6.3 *Decision Trees and Random Forests*

A decision tree (DT) is a non-parametric statistical learning model that is commonly being used for both classification and regression problems. The model, which is mostly represented by a binary tree, divides the training data into several subspaces. A node of the tree, e.g., v, indicates a subspace $X_v \in X$ and the root node denotes the whole input space X. Each non-leaf node v has a split value s that is used to divide X_v into two disjoint subspaces, e.g., X_{v_l} and X_{v_r}. The input features correspond to the nodes in the tree.

A random forest is an ensemble extension of DTs in which several DTs are built by bootstrap sampling of a randomly selected subset of input features. A majority vote over the outcomes of the individual decision trees is used to generate the outcome of the RF. For every node in a tree, the regression cost function R_c is defined as follows:

$$R_c(v) = \frac{1}{n} \sum_{x_{k,*} \in v} (y^k - \bar{y}(v))^2 \tag{4.11}$$

where $\bar{y}(v)$ is the average output value at the node v. When v is split into two children, v_l and v_r, the total cost at node v is the upper bound of the sum of its children's costs:

$$
\begin{aligned}
R_c(v) &= \frac{1}{n} \sum_{x_{k,*} \in v} (y^k - \bar{y}(v))^2 \\
&= \frac{1}{n} \sum_{x_{k,*} \in v_l} (y^k - \bar{y}(v))^2 + \frac{1}{n} \sum_{x_{k,*} \in v_r} (y^k - \bar{y}(v))^2 \\
&\geqslant \frac{1}{n} \sum_{x_{k,*} \in v_l} (y^k - \bar{y}(v_l))^2 + \frac{1}{n} \sum_{x_{k,*} \in v_r} (y^k - \bar{y}(v_r))^2 \\
&\geqslant R_c(v_l) + R_c(v_r)
\end{aligned}
\tag{4.12}
$$

where v_l and v_r, are the left and right child nodes of v.

Finding the smallest tree that minimizes the regression cost for a given dataset is an NP-complete task. Thus, a greedy strategy is used to keep the regression tree small. The strategy looks for the best split of an internal node v in a way that locally minimizes the regression cost of its children, v_l and v_r. To that end, an impurity measure is defined for every internal node v as:

$$
I_m(v) = R_c(v) - R_c(v_l) - R_c(v_r) \tag{4.13}
$$

The largest impurity is then selected to split the node v as:

$$
\underset{s}{\operatorname{argmax}} \Big(I_m(v) \Big) \tag{4.14}
$$

where s denotes the split value of node v. In this way, the average of the obtained impurities for every feature in each decision tree can be utilized as a measure of feature ranking as follows:

$$
r(\mathbf{x}_i) = \frac{1}{n_t} \sum_{v \in S_i} I_m(v) \tag{4.15}
$$

where n_t is the number of decision trees in the random forest, and S_i indicates the set of nodes split by feature x_i. In fact, the obtained ranking scores can be interpreted as the relevancy of each feature with respect to the given output variable. Unlike the previous parametric methods, RFs do not provide any associated parameter w_i to each feature x_i; instead the importance of the features is measured via Equation (4.15).

With a minor modification, RFs can be exploited for the problem of multi-output feature selection. In this regard, the information gain is used as the splitting criteria and is modified to compute the average reduction of impurities for all output variables. This requires storing all output values in the leaves of each decision tree.

4.6.4 *Comparison of the Feature Ranking Methods*

This section briefly compares the described feature ranking methods LASSO, HSIC LASSO, and random forest with respect to the requirements discussed.

The first requirement, the independence of distribution of sample data, is satisfied for all three methods. The second requirement, the ability to deal with linear and non-linear correlations, is fulfilled by HSIC LASSO and RF, where HSIC LASSO can even deal with non-linear correlations between features and output(s) separately. LASSO is weaker as it is limited to linear correlations. The third requirement is that the methods should be applicable to multi-output functions and, again, can be met by all three methods. LASSO and HSIC LASSO require an aggregation function, and RF needs a minor modification to be able to handle multi-output functions. The last requirement is that the training time should be negligible in comparison with the path planning and the binary search phases of jump search, which we demonstrate experimentally in Section 4.9.4.

4.7 DETERMINING THE FIGURE-OF-MERIT

Jump search's path planning phase relies on a heuristic function, denoted as figure-of-merit (FoM), which includes the candidates' impact factors for area and error to rank the direct successors of a node in the search path. We provide and experiment with two FoM functions, $\mathrm{FoM}_v(s, s_s, c)$ that is applied when the candidates' area impact factors depend on their local WC errors, and $\mathrm{FoM}_c(s, s_s, c)$ for use with a constant area impact factor per candidate.

The set S holds all successor nodes of node s, and a node's direct successor $s_s \in S$ distinguishes in exactly one candidate, i.e., the error $\epsilon(s_s(c))$ of candidate c of the successor node $s_s \in S$ is different from the candidate's error $\epsilon(s(c))$ of node s. Consequently, for the FoMs, it is sufficient to only evaluate the differing candidate of the nodes.

The heuristic function FoM_v that considers a function for the area impact factor if_{area} is defined as follows:

$$\mathrm{FoM}_v(s, s_s, c) = \frac{\Delta A_v(s, s_s, c)}{\Delta E(s, s_s, c)} \times \frac{1}{if_{err}(c)+1} \tag{4.16}$$

$$\Delta A_v(s, s_s, c) = if_{area}(c, \epsilon(s_s(c)) - if_{area}(c, \epsilon(s(c)) \tag{4.17}$$

$$\Delta E(s, s_s, c) = \epsilon(s_s(c)) - \epsilon(s(c)) \tag{4.18}$$

This function computes the FoM of the successor node s_s of node s by evaluating the merit resulting from increasing the error of candidate c. With ΔA_v, the FoM considers the difference in the area impact factor for the given local quality constraints (or local errors) $\epsilon(s_s(c))$ and $\epsilon(s(c))$ of candidate c in node s_s or s, respectively (cf. Section 4.5). ΔE accounts for the difference in the error between the two nodes, which is needed if the changes in error are non-uniform, as it is the case for precision scaling, where the error changes are described by the power of two. Hence, the term $\frac{\Delta A}{\Delta E}$ essentially

describes the slope of the change in the area over the error that is, in turn, further adjusted by the term $\frac{1}{if_{err}(c)+1}$, i.e., the candidate's contribution to the overall circuit error (cf. Section 4.6). Since $if_{area}(c)$ and $if_{err}(c)$ are both in $[0, 1]$, Equation (4.16) adds a constant to the denominator, which results in a range of $[0, \frac{1}{2}]$ for FoM_v. Among all successor nodes, the successor with the maximum $FoM_v(s, s_s, c)$ is then selected as the next node on the search path (cf. Line 13).

The heuristic function FoM_c that considers a constant area impact factor is defined as follows:

$$FoM_c(s, s_s, c) = \frac{\Delta A_c(s,s_s,c)}{\Delta E(s,s_s,c)} \times \frac{if_{area}(c)}{if_{err}(c)+1} \tag{4.19}$$

$$\Delta A_c(s, s_s, c) = \left(\frac{\epsilon(s_s(c))}{\epsilon_{Max}(c)}\right)^k - \left(\frac{\epsilon(s(c))}{\epsilon_{Max}(c)}\right)^k \tag{4.20}$$

Again, the heuristic function evaluates all successor nodes $s_s \in S$ of the current node s in the search path by computing the gains through modifying candidate c. While using a constant area impact factor for a candidate requires only one additional synthesis step per candidate during pre-processing, it also turns the second term into a constant for a candidate and, thus, statically orders all candidates. In order to base the search also on the actual error, we introduce ΔA_c in the first term that emphasizes changes around small relative local errors. Additionally, in ΔA, the exponent k helps balancing the impact of the local errors. For example, $k = 1$ results in a linear dependency of FoM_c from the candidate's LQC, which, in turn, again leads to a static ordering of the candidates. Choosing $k < 1$, on the other hand, emphasizes the impact of the local error in the sense that FoM_c decreases faster with increasing local error, possibly allowing the search to select other configurations with less aggressively approximated candidates. In this dissertation, we employ $k = 0.5$. FoM_c also uses ΔE to estimate a slope and account for non-uniform changes in the error, and, again, FoM_c is in the range $[0, \frac{1}{2}]$.

4.8 IMPLEMENTATION OF JUMP SEARCH

Figure 4.5 shows the block diagram of jump search's implementation. The pre-processing step has been implemented in an automated flow using custom Python scripts; in fact, the implementation was part of the outcome of a Master's thesis [21] that I have supervised. The CIRCA framework (see Chapter 3) implements the execution phases path planning and binary search. Figure 4.5 highlights the part of CIRCA which has been modified for jump search's integration, i.e., the search space exploration block in the quality assurance, approximation, estimation, and search space exploration (QUAES) stage.

The impact factors are computed independently of each other as well as of the rest of the flow. Through this modularity, new impact factors can be considered and different methods can be employed to compute of the individual factors. The computed impact factors are stored as parameters for

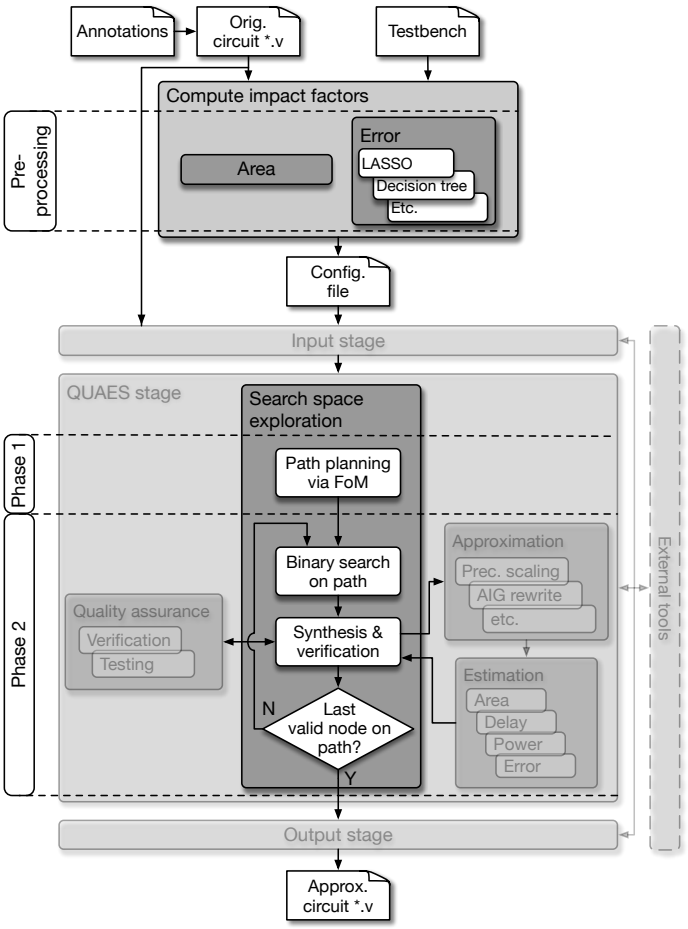


Figure 4.5: Overview of the jump search in the CIRCA framework [101]. Highlighting indicates modified parts of CIRCA.

jump search in CIRCA's configuration file, which is used to set up the approximation process. Jump search's path planning has been implemented as a setup phase in CIRCA's QUAES stage, and binary search follows CIRCA's suggested data flow, which allows for a swift exchange of approximation methods, the use of different error and target metrics, and/or target technologies. Finally, the resulting AxC is given to the user in the form of a Verilog design file.

As a reference for our experiments, we have decided on CIRCA's simulated annealing and hill climbing algorithms since they are commonly used [69, 94, 101] and previously performed experiments have shown that the algorithms achieve significant area savings in different runtimes (cf. Section 3.5). Simulated annealing forms a compromise between a greedy search and a random walk. The simulated annealing algorithm starts from the original circuit and is only allowed to progress towards increasing approximations. In this way, simulated annealing is restrained from being too explorative and follows our assumption of higher savings with higher degrees of approximation. As a result, simulated annealing iteratively progresses a random path through the search space. However, compared to random walk, simulated annealing

might reject a bad move and decide on a better one. A move that improves the target metric is always accepted.

4.9 EXPERIMENTAL EVALUATION

In this section, we will discuss our experimental results. We performed several experiments to evaluate jump search and compare it against the commonly used search algorithms simulated annealing and hill climbing.

4.9.1 *Experimental Setup*

For the evaluation of our approach, we have used seven circuits from the approximate computing benchmark set PaderBench. Table 4.1 shows the benchmark circuits, the number of candidates for each benchmark, the search space's size determined via Equation (4.1), and the path length for jump search determined via Equation (4.2). Comparing the size of the search spaces to the number of selected AxCs for the paths shows directly that JS significantly reduces the number of AxCs-of-interest.

Table 4.1: Overview of the benchmarks.

Benchmark	#Candidates	Search space size	Path length
basic_sad	16	5.63×10^{15}	156
butterfly	5	1.68×10^{7}	145
fir_pipe_16	22	9.21×10^{18}	397
pipeline_add	2	441	43
rgb2ycbcr	12	1.81×10^{16}	273
ternary_sum_nine	3	116,640	55
weight_calculator	4	28,561	53

Our experiments have been performed on the OCuLUS compute cluster of the Paderborn Center for Parallel Computing (PC²), which runs Scientific Linux 7.2 (Nitrogen) and comprises nodes with an Intel® Xeon E5-2670@2.6GHz (16 cores). For each job, eight Gigabyte of main memory has been provided. Simulated annealing and hill climbing have been used as reference implementations (see Witschen et al. [101] and Section 4.4). Due to the involved randomness in the search methods (simulated annealing (SA) involves random acceptances and hill climbing (HC) uses randomness as a tie-breaker), we ran each experiment five times and report the averaged results. We invoked ABC [14] to report on the circuit's FPGA 4-LUT usage via the *if* command and ABC's *dprove* command to formally verify the quality of the AxCs. A single verification has a time limit of seven hours; exceeding the time limits deems the AxC invalid.

For JS, we first simulated each circuit with a dataset of 10,000 random vectors and sampled the output values of the candidates as well as the circuit's output. The benchmark circuit `basic_sad` is part of an image processing sys-

tem and a 512×512-pixel image has been used to simulate the circuit. From the sampled data, the feature ranking and selection techniques discussed in Section 4.6 are used to determine the $\mathtt{if}_{\mathtt{err}}$ of each candidate. Namely, the employed techniques are random forest, LASSO, and HSIC LASSO. Secondly, for each candidate, two $\mathtt{if}_{\mathtt{area}}$s are determined with the methods detailed in Section 4.5, i.e., a constant impact factor and an impact factor function. For determining the $\mathtt{if}_{\mathtt{area}}$ via curve fitting, five data points (the original circuit plus four approximated designs, cf. Section 4.5) have been sampled for each candidate to keep the number of synthesis steps small.

We have applied the worst-case error metric for each benchmark circuit, varying the error bound from 0.50% to 5.00% of the maximal possible output value, and performed several experiments. In the following, we will first compare summarized area and runtime results of jump search to the results achieved with simulated annealing and hill climbing in Section 4.9.2. Secondly, Section 4.9.3 elaborates on the number of performed verifications and syntheses during ALS for the three approaches. Section 4.9.4 then details the pre-processing phase of jump search and discusses the workload that the different methods for determining the impact factors create. Finally, we discuss the impact of the different figure-of-merits and feature ranking methods on jump search's outcome in Section 4.9.5.

4.9.2 *Experimental Evaluation of Jump Search*

Figure 4.6 summarizes the experimental results of JS. The figure shows the achieved relative area on the left-hand side and the runtimes on the right-hand side. Note that the runtime of the pre-processing phase for jump search is excluded from the runtimes and is discussed separately in Section 4.9.4. The green bars indicate the results for jump search, the red bars for simulated annealing, and the blue bars for hill climbing. For jump search, the figure reports the averaged results over all combinations of the three feature ranking methods, i.e., LASSO, HSIC LASSO, and RF, and the two figure-of-merits, i.e., $\mathrm{FoM}_{\mathrm{c}}$ and $\mathrm{FoM}_{\mathrm{v}}$. Additionally, all experiments' minimal and maximal values are visualized through black bars with caps.

For JS, we observe area reductions for all benchmarks with negligible variations among the different approaches of jump search, indicated by small black bars (see Section 4.9.5 for more details). In fact, JS achieves either comparable or better results than SA or HC. For the benchmark `rgb2ycbcr`, for example, JS achieves up to $\approx 18\%$ higher reductions in area compared to SA and up to $\approx 37\%$ compared to HC. Merely, for the benchmark `pipeline_add`, SA achieves more prominent area reductions, which, however, are not larger than $\approx 4.5\%$. Neither of the approaches achieves significant savings for the benchmarks that are difficult to approximate, i.e., `butterfly`, `fir_pipe_16`, and `weight_calculator`.

The resulting runtimes of the three approaches differ significantly. In all experiments, JS is faster than simulated annealing and achieves significant speed-ups of up to $\approx 58\times$ (cf. `butterfly`, WC error 0.5%), thus, reduc-

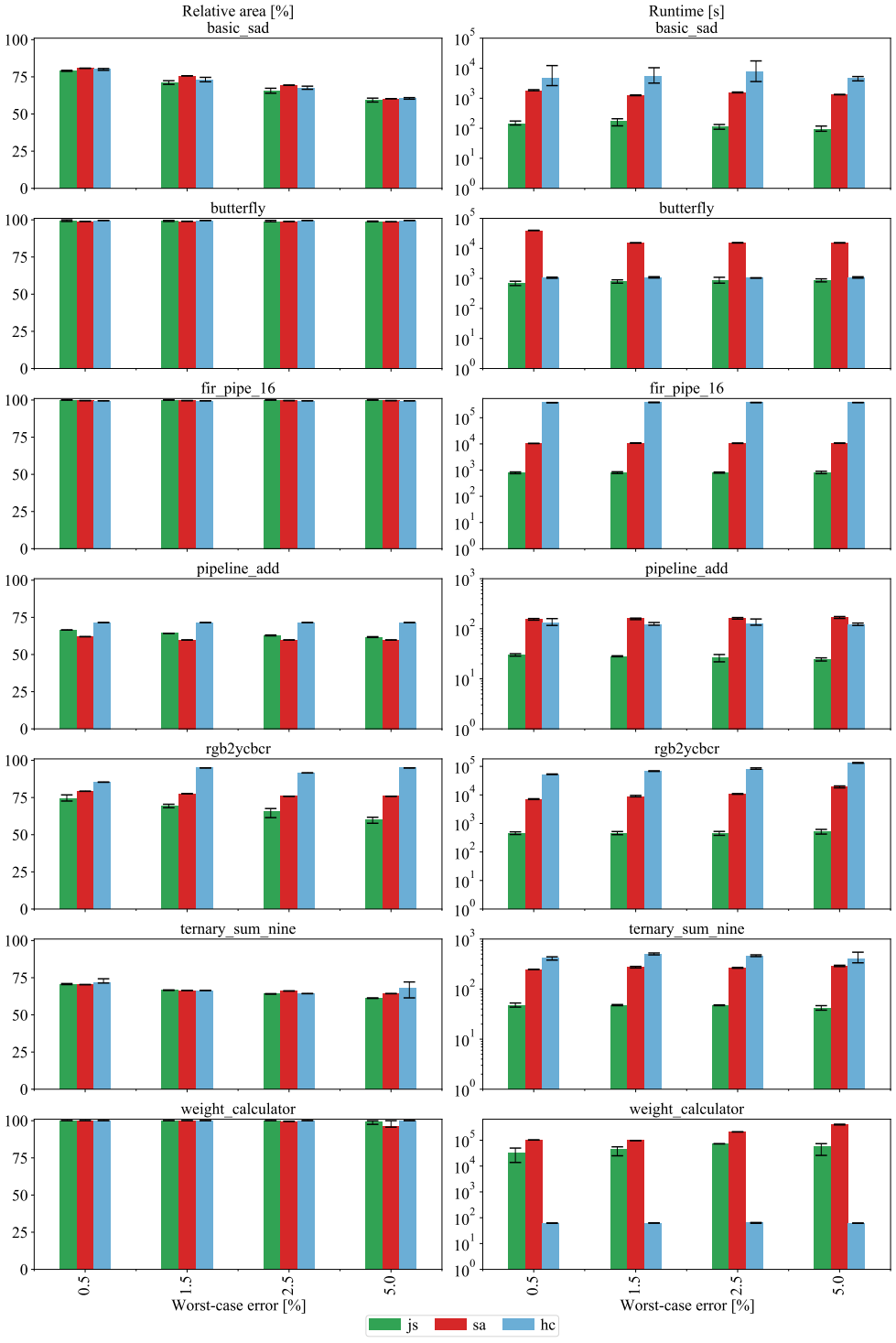


Figure 4.6: Average relative area and timing. The black bars with caps indicate the minimums and maximums from all experiments.

ing the runtime by over an order of magnitude. Compared to hill climbing, JS achieves even more prominent speed-ups of up to $\approx 468\times$ for the benchmark `fir_pipe_16` with a worst-case error of 0.5%. The benchmark `weight_calculator` states an exception where HC achieves very low runtimes due to terminating after the first iteration since the search gets stuck in a local minimum immediately.

4.9.3 *Comparison of Synthesis and Verification Steps*

Figure 4.7 shows the average number of verifications on the left-hand side and the average number of syntheses on the right-hand side for the three approaches. As discussed in Section 4.4, JS performs significantly fewer verifications and synthesis steps compared to SA or HC, which, in turn, results in the significantly reduced runtimes shown in the previous section. In fact, JS generally reduces the number of verifications and syntheses by one order of magnitude. As discussed in the previous section, however, the benchmark `weight_calculator` marks an exception for HC as the search gets stuck in a local minimum immediately; the effect can be observed through the number of performed verifications for this benchmark in Figure 4.7. While syntheses are performed, verifications of AxCs are not performed by HC – apart from the original design.

Furthermore, the benchmark highlights the impact of formal verification on the runtime. Even though only a few verifications are performed with JS or SA, the runtimes increase significantly compared to HC (cf. Figure 4.6). In fact, the verifications are terminated by exceeding the time limit of seven hours rather than by the formal verification engine deciding the AxC's validity.

For the benchmark `butterfly`, the runtimes of JS are slightly shorter than with HC (cf. Figure 4.6). While JS performs a few more verifications, HC performs an order of magnitude more synthesis steps, which leads to the increased runtimes in this case, and again exemplifies the impact of the verification on the overall runtime.

4.9.4 *Evaluation of the Pre-processing Phase*

The runtimes of jump search in Figure 4.6 do not consider the runtime of the pre-processing phase. Thus, to determine the total runtimes of JS, we investigate the runtime of determining the impact factors $\mathtt{if}_{\mathtt{area}}$ and $\mathtt{if}_{\mathtt{err}}$. Note that both impact factors can be computed independently of each other, and the pre-processing phase can thus be parallelized. Consequently, the longest-running task dictates the runtime of the pre-processing phase.

Table 4.2 shows the runtimes for determining $\mathtt{if}_{\mathtt{area}}$. The original circuit has to be synthesized once to establish the baseline, and the runtime is shown in the column *Synthesis orig.* of the table. The table also lists the number of candidates in each benchmark circuit, which, in turn, influences the number of additional synthesis steps. The overhead introduced by determining the constant $\mathtt{if}_{\mathtt{area}}$s is listed in column *Synthesis 1x/cand*. The

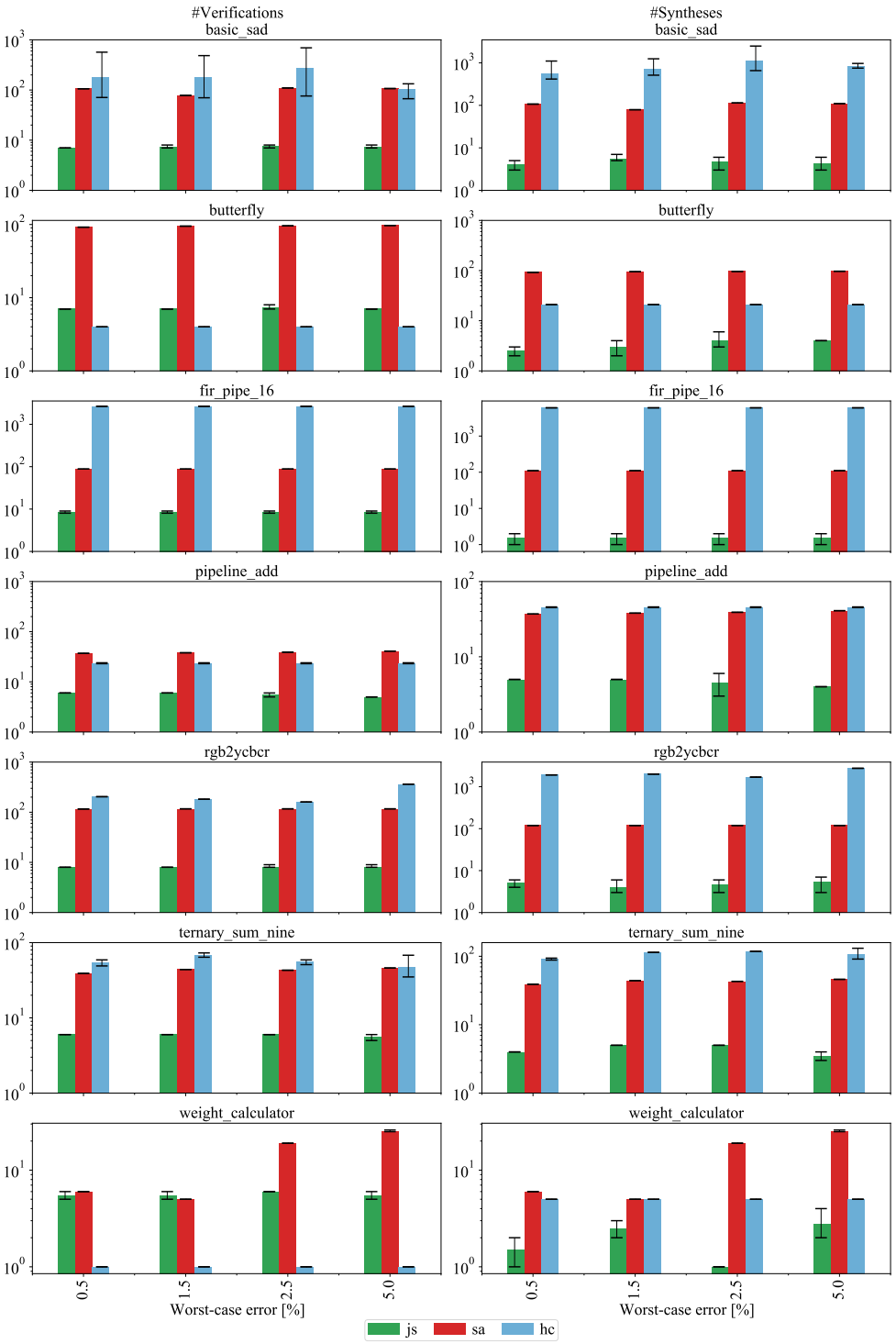


Figure 4.7: Average number of verifications and syntheses.

column shows the runtime for synthesizing the circuit candidate-many times following the methodology described in Section 4.5. For example, the benchmark `basic_sad` has 16 candidates; hence, 16 additional synthesis steps are required to determine the constant $\texttt{if}_{\texttt{area}}$ for each candidate in the circuit. The total runtime for sampling the four data points for each candidate (column *Synthesis 4x/cand.*) and the curve fitting (column *Curve fitting*) indicate the overhead for determining the $\texttt{if}_{\texttt{area}}$ functions. In the example of the circuit `basic_sad`, four additional synthesis steps are performed to sample the data points for each candidate, resulting in a total of 16×4 additional synthesis steps. Note that the overhead from the additional synthesis steps increases linearly with the number of candidates or the number of sampled data points, respectively. As the table shows, the runtimes for determining $\texttt{if}_{\texttt{area}}$ are relatively small compared to the runtimes of the ALS process and add no significant overhead that causes JS to endure longer total runtimes than the reference implementations – even though pre-processing may take several minutes (cf. `fir_pipe_16`).

Table 4.2: Runtimes for determining $\texttt{if}_{\texttt{area}}$. The number of candidates present in each benchmark circuit and the runtime for synthesizing the original benchmark circuit is listed. Along the total runtimes for synthesizing the circuit once per candidate and four times per candidate, the runtime for curve fitting is shown.

Benchmark	**#Candidates**	**Synthesis**†		**Curve fitting** [s]
		orig.	**1x / candidate**	**4x / candidate**
basic_sad	16	1.70	29.59	2:04.04	2.44
butterfly	5	10.29	43.05	3:30.10	1.72
fir_pipe_16	22	8.02	2:47.38	11:32.08	2.81
pipeline_add	2	0.50	0.84	3.47	1.70
rgb2ycbcr	12	4.97	53.11	3:45.04	2.44
ternary_sum_nine	3	0.78	2.28	10.35	1.83
weight_calculator	4	2.52	10.71	41.75	1.82

† The runtimes are shown in the format minutes:seconds.milliseconds.

Table 4.3 shows the runtimes of the feature ranking methods used to determine the impact factors $\texttt{if}_{\texttt{err}}$. Overall, LASSO shows the lowest runtimes among the three methods, followed by RF. HSIC LASSO is the most complex method among the approaches, and thus, endures the longest runtimes. Compared to the ALS process, the runtimes of the feature ranking methods are again low and are comparable to the runtimes of determining the impact factors $\texttt{if}_{\texttt{area}}$, thus, adding no substantial overhead to the process.

4.9.5 *Discussion on the Figure-of-merits and Feature Ranking Methods*

The achieved area savings in Figure 4.6 indicate that the differences between the employed FoM or the utilized feature ranking method are marginal for our benchmark set. In fact, JS indicates variations in the results only for a

Table 4.3: Runtimes of the feature ranking methods for determining $\mathtt{if}_{\mathtt{err}}$.

Benchmark	Random forest	LASSO	HSIC LASSO
basic_sad	36.82	5.05	55.43
butterfly	9.68	2.24	22.90
fir_pipe_16	18.27	2.89	35.68
pipeline_add	4.15	2.21	10.44
rgb2ycbcr	11.46	1.05	37.27
ternary_sum_nine	4.68	1.44	11.91
weight_calculator	5.49	1.52	13.81

few benchmarks, and thus, this section provides a more detailed discussion on the experiments that aggregated to the average results shown in Section 4.9.2. Figure 4.8 shows the area results for the benchmarks `basic_sad`, `butterfly`, `pipeline_add`, and `rgb2ycbcr`. For the six different combinations of the feature ranking method and FoM, the figure shows the achieved normalized area. The colors indicate the utilized feature ranking method and the saturation the FoM.

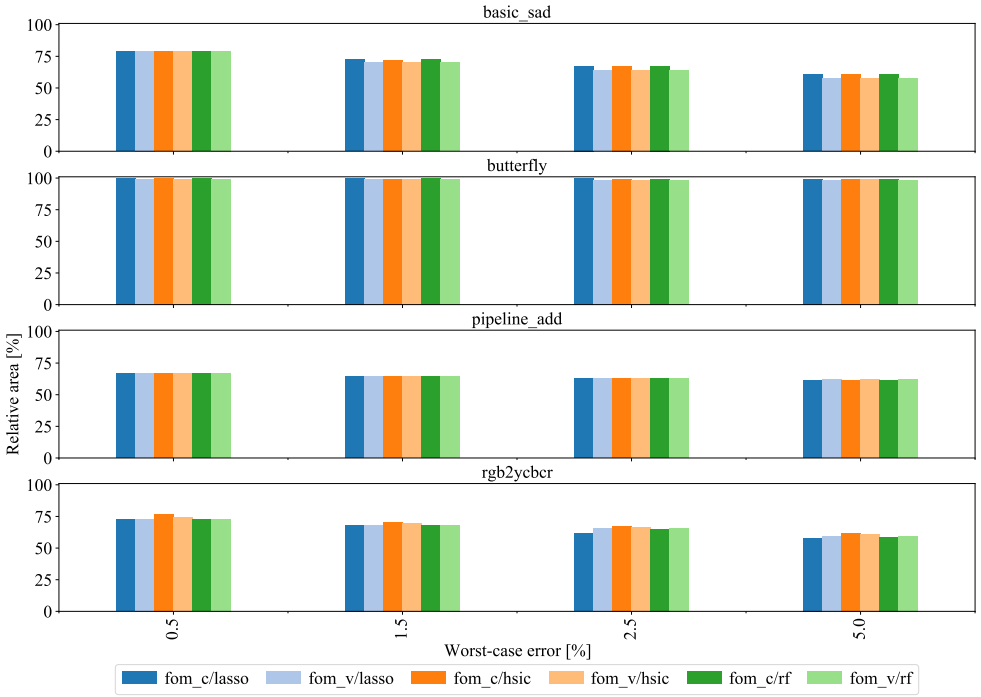


Figure 4.8: Normalized area for different combinations of feature ranking method and figure-of-merit.

For the benchmarks `butterfly` and `pipeline_add`, no significant differences between the methods can be observed. The benchmark `pipeline_add` provides only two candidates for which all feature ranking methods determine the same impact factors. Consequently, all methods achieve the same savings, and differences can only be observed between the different FoMs,

which are $< 1\%$. For `butterfly`, this is likely caused by the marginal savings that can be achieved.

The benchmarks `basic_sad` and `rgb2ycbcr` show more prominent variations in comparison. Among the feature ranking methods, LASSO provides the best results since it performs best for the benchmark `rgb2ycbcr`; for `basic_sad`, no differences can be seen between the three ranking methods. While the FoM_v, which uses a function for $\texttt{if}_{area}$, achieves better results for `basic_sad` than the FoM with a constant $\texttt{if}_{area}$, FoM_c, the situation inverses for `rgb2ycbcr`, where FoM_c in combination with LASSO achieves the best results.

Table 4.4: Overview of the best combinations of figure-of-merit and feature ranking method for each benchmark and worst-case error bound. An $*$ indicates that all approaches achieved the best result.

Benchmark	**Worst-case error** [%]							
	0.5		**1.0**		**2.5**		**5.0**	
	FoM	**Feat.**	**FoM**	**Feat.**	**FoM**	**Feat.**	**FoM**	**Feat.**
basic_sad	FoM_c	$*$	FoM_v	$*$	FoM_v	$*$	FoM_v	$*$
butterfly	FoM_v	$*$	FoM_v	$*$	FoM_v	$*$	FoM_v FoM_v	LASSO RF
fir_pipe_16	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
pipeline_add	$*$	$*$	$*$	$*$	FoM_c	$*$	FoM_c	$*$
rgb2ycbcr	FoM_c FoM_c	LASSO RF	FoM_c	RF	FoM_c	LASSO	FoM_c	LASSO
ternary_sum_nine	$*$ $*$	LASSO RF	$*$	HSIC	$*$ $*$	LASSO RF	$*$ $*$	LASSO RF
weight_calculator	$*$	$*$	$*$	$*$	$*$	$*$	FoM_v FoM_v	LASSO RF

Table 4.5: Summary of the number of times a combination of figure-of-merit and feature ranking method achieved the best result. The first row or column indicates the number for the respective method in that column or row, respectively. The values in the matrix represent the combination of the figure-of-merit and feature ranking method.

		LASSO	**HSIC LASSO**	**Random forest**
		26*	19	25
FoM$_c$	20	18	13	17
FoM$_v$	**21***	**20***	16	**20***

* Best-performing method or combination of methods.

Table 4.4 shows the best-performing FoM and feature ranking method for each benchmark and WC error bound; an $*$ indicates that all FoMs or ranking methods achieved the best result. Finally, Table 4.5 summarizes the results and shows how often each FoM, feature ranking method, or

their combinations lead to the best result. Table 4.4 reveals that the FoM makes the difference in most cases rather than the employed ranking method. Furthermore, from the two tables, it can be seen that LASSO and RF most often achieve the best result, usually in combination with an if_{area} function in the FoM_{ν}.

In summary of our experimental results, we conclude that determining a function for the impact factor if_{area} is beneficial since the area savings are generally improved while the introduced overhead is marginal compared to overall runtimes. However, against our expectations, the non-linear feature ranking method HSIC LASSO has not led to better results than the linear technique LASSO for our benchmarks. An explanation for this outcome may be the relatively small number of candidates in our benchmark circuits, for which a simple yet efficient method such as LASSO suffices. Feature ranking methods generally deal with a significantly larger number of candidates where differences may become more prominent and, in these cases, HSIC LASSO may perform better than LASSO. However, the formal verification is currently a bottleneck and limits the number of candidates in our setup. Moving towards designs with substantially more candidates would lead to prohibitively long runtimes and, thus, is out of reach with our current resources.

4.10 CONCLUSION

In this chapter, we have presented the jump search, a fast method for synthesizing approximate circuits. Jump search divides into a pre-processing phase, a path planning phase, and a binary search to find a suitable AxC rapidly by omitting synthesis and verification wherever possible. During the pre-processing phase, jump search determines two impact factors for each candidate that represent the candidate's impact on the area and the circuit error. Incorporating the impact factors into the figure-of-merit, jump search evaluates AxCs and plans a path through the search space without referring to costly synthesis or verification steps. Finally, jump search employs a binary search to find the deepest AxC on the planned path in order to achieve significant savings.

We have analyzed different methods for determining the impact factors during pre-processing and discussed two figure-of-merits for path planning. In addition, we have presented approaches to determine a constant if_{area} or a if_{area} function via curve fitting. Finally, for $\mathrm{if}_{\mathrm{err}}$, we analyzed and compared three different feature ranking methods, namely LASSO, HSIC LASSO, and random forest.

Our experimental results confirmed that jump search leads to significant improvements in the target metric hardware area, which are comparable to the commonly used search methods simulated annealing and hill climbing. However, jump search's runtimes are substantially lower. In fact, through the significantly reduced number of verifications and syntheses, we have achieved speed-ups of up to $\approx 468\times$ without sacrificing reductions in area.

5

MUSCAT: A MUS-BASED CIRCUIT APPROXIMATION TECHNIQUE

5.1 OVERVIEW

This dissertation considers search-based approximate logic synthesis (ALS), and this chapter devotes itself to the approximation step of the general ALS presented in Section 2.1.1. The approximation step receives as input a candidate circuit and applies approximations according to a set of parameters dictated by the overarching search-based ALS (cf. Section 3.6 and Section 2.1.2). Thus, the approximation step acts as an executive body and applies approximations to the candidate circuit, following the defined methodology of the employed ALS technique.

To further structure the ALS in the approximation step, we follow Scarabottolo et al. [75] and distinguish between three categories for ALS methods (see Sections 2.1.2 to 2.1.4): approximate high-level synthesis (AHLS), Boolean rewriting, and netlist transformation. Furthermore, in their survey, Scarabottolo et al. describe three components of a general ALS technique in the approximation step: error modeling, the actual ALS method, and quality-of-result evaluation (also called *quality assurance* or *quality verification*).

The error model is created for a specific input design and used by the actual ALS method to quickly estimate the impact of an approximation on the overall circuit error. Quality verification is needed to verify that an approximate circuit (AxC) is valid, i.e., adheres to the quality constraints. While error models can benefit the approximate outcome [77], an open issue is that these models are often inexact and too conservative [75], leading to unused approximation potential (cf. Section 2.1.3). Furthermore, a dedicated quality verification applied from time to time may force the ALS to back-track and revoke approximations to establish a valid AxC.

In this chapter, we present MUSCAT, a minimal unsatisfiable subset (MUS)-based circuit approximation technique. MUSCAT is a netlist transformation technique that creates AxCs *valid-by-construction* regarding their quality constraints, and thus, neither requires error models nor dedicated quality verification.

Our ALS technique is based on the commonly-used concept of gate level pruning (GLP) that substitutes connections between gates by constant values (see Section 2.1.4). To achieve this, MUSCAT firstly augments an input netlist by so-called *cutpoints*, which, if activated, perform the substitution.

Then, MUSCAT employs the novel approach of utilizing formal verification engines to determine MUSes. The MUSes identify the maximal number of cutpoints that can be activated together safely, i.e., the constructed AxC is guaranteed to adhere to the quality constraint. Activating any further cutpoints would render the AxC invalid. Hence, a MUS specifies an optimal solution w.r.t. the number of activated cutpoints. Our practical experiments show that after applying standard logic synthesis tools to subsequently minimize the circuit, we indeed achieve significant improvements over state-of-the-art techniques. Since formal methods are often questioned, it is important to note that they have matured significantly in the recent past and can verify designs with millions of gates and hundreds of inputs in a reasonable time [89, 90].

Comparing MUSCAT to related work reveals that our MUS-based approach applies approximations via netlist transformations similar to GLP [77, 78] and also resembles the concept of cuts used in the Boolean rewriting technique AIG rewriting [22] (see Section 2.1.3). However, our approach is fundamentally different from related work in netlist transformation and most approaches in Boolean rewriting since we generate AxCs that are valid-by-construction without the need for an error model or dedicated quality verification steps. The independence from an error estimation or propagation model enables our approach to support any non-statistical error metric, while GLP and its successors only provide an error model for the worst-case (WC) error. Furthermore, AIG rewriting considers only nodes on the critical paths – usually carry-chains in arithmetic components – making the approach often too aggressive (see [101] and Section 5.3). Comparing MUSCAT to the Boolean rewriting technique SALSA [91] (see Section 2.1.3), which also spares error modeling and dedicated quality verification steps, we identify two major differences in the employed ALS technique and the abstraction level. MUSCAT inserts cutpoints into a netlist and computes minimal unsatisfiable subsets, while SALSA operates on a more abstract level as a Boolean rewriting approach and relies on standard don't care optimization.

In summary, our contributions are:

- We propose MUSCAT, a novel MUS-based ALS technique that exploits modern formal verification engines to generate AxCs with quality constraints valid-by-construction. Our technique does not require an error model and thus overcomes limitations of previous work; in particular, it allows for supporting any non-statistical error metric.
- We present experiments showing that, compared to state-of-the-art ALS, our method achieves up to 80% higher area savings.
- MUSCAT is open-source and publicly available[1].

My colleague, Tobias Wiersema, supported me developing MUSCAT's concept, and my student research assistant, Matthias Artmann, helped with the implementation. We have presented MUSCAT in our conference publication [106], and follow and cite its discussions largely in Sections 5.1 and 5.2.

1 https://git.uni-paderborn.de/muscat/muscat

The remainder of this chapter structures as follows: Section 5.2 presents our ALS technique, and Section 5.3 discusses various experiments and results achieved with MUSCAT that we have partially presented in our conference publication. A case study in Section 5.4 evaluates MUSCAT on different levels of abstraction. Finally, Section 5.5 concludes the chapter.

5.2 METHODOLOGY

We represent the gate level netlist of a combinational circuit as a directed acyclic graph (DAG) $G(N, E)$, where $n \in N$ represents a node (gate) in G, and $(a, b) \in E$ represents an edge (a connection between gates). For an edge (a, b), node a's output drives node b's input. We augment a given DAG G with so-called *cutpoints*. A cutpoint is inserted at an edge (a, b) and consists of one or two gates by which we can either *activate* or *deactivate* the cutpoint. An *activated* cutpoint ruptures the connection and drives b's input by either constant 0 or 1; a *deactivated* cutpoint leaves the connection unchanged, i.e., the original signal from node a propagates to node b.

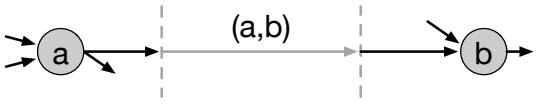
For a given DAG G, we denote the set of all cutpoints as C. At design time, a *cutpoint configuration* (short: *configuration*) defines whether each cutpoint $c \in C$ is activated or deactivated. Activated cutpoints offer optimization potential, and subsequent logic synthesis tools will remove dangling nodes and simplify the logic through constant propagation. We exploit modern formal verification techniques to determine an optimal configuration via *minimal unsatisfiable subsets*. A MUS provides an optimal configuration in terms of the number of activated cutpoints and, furthermore, enables us to construct AxCs valid-by-construction.

We first discuss different cutpoint designs and the approximation miter for determining and verifying configurations in the following. Then, we elaborate on minimal unsatisfiable subsets, and, finally, we present our ALS algorithm and discuss the cutpoint insertion.

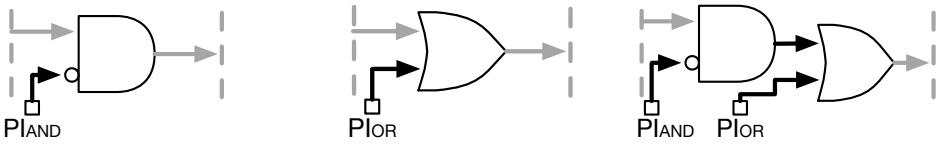
5.2.1 *Cutpoints*

Figure 5.1a shows a segment from a graph $G(N, E)$ with the nodes $a, b \in N$ and the edge $(a, b) \in E$. The gray, vertically-dashed lines indicate the insertion of a cutpoint $c \in C$ for the edge (a, b). Figures 5.1b to 5.1d present different cutpoint designs.

The AND-cut in the edge (a, b), as shown in Figure 5.1b, comprises an AND gate with one input connected to the output of a and the second one exposed as inverted new primary input (PI) PI_{AND}. Depending on the value for this new primary input, the edge remains either unchanged or is cut with $PI_{AND} = 1$, and the logic value 0 drives the input of node b. In this way, the activated cutpoint provides potential for optimization since the logic driving node a's output becomes obsolete and the subsequent logic simplifies through constant propagation.



(a) Original edge in the graph.



(b) AND-cut. (c) OR-cut. (d) AND-OR-cut.

Figure 5.1: Concept of replacing an edge with a cutpoint in a netlist. Dashed gray lines indicate the cut, solid gray lines indicate the original connection, and boxes represent newly added primary inputs. Taken from [106].

The OR-cut shown in Figure 5.1c inverses the AND-cut's behavior, i.e., propagates a constant 1 to the input of node b when activated via $PI_{OR} = 1$. The AND-OR-cut displayed in Figure 5.1d allows for propagating either the constant 0 or 1 when activated but exposes two new primary inputs, PI_{AND} and PI_{OR}, to realize the cutpoint. While the AND-cut and the OR-cut are alternatives that, in general, will create different optimization potential for subsequent logic synthesis, the AND-OR-cut subsumes the functionality of both single gate cuts at the cost of increased complexity through doubling the number of newly exposed PIs.

Conceptually, cutpoints can be added to every edge in a graph. However, reducing the number of cutpoints can reduce complexity and thus be beneficial to shorten the runtime of ALS. Section 5.2.4 discusses different heuristics that allow the evaluation of the potential merit of a cutpoint in the graph.

5.2.2 *Approximation Miter*

Our ALS technique determines a suitable configuration for a given set of cutpoints C and simultaneously verifies the validity of the resulting AxC by formulating a satisfiability (SAT) problem in the satisfiability modulo theories (SMT) domain (see Section 2.2). To that end, we construct an approximation miter as shown in the example of Figure 5.2 for a circuit with six nodes, five primary inputs, three primary outputs (POs), and a set C of 16 possible cuts. The approximation miter comprises the exact circuit, the AxC including the cutpoints, and verification logic. The verification logic determines the circuit's error ϵ by comparing the POs of the AxC with the POs of the exact circuit and checking whether ϵ exceeds a given threshold T that corresponds to the specified quality constraint. If the chosen configuration of cutpoints leads to an error exceeding T, the miter becomes satisfiable (short: SAT) and the output of the approximation miter will be raised, indicating a quality constraint violation. Consequently, if we prove the unsatisfiability (UNSAT) of

the approximation miter, we have a guarantee that the chosen configuration actually leads to an AxC adhering to the quality constraint.

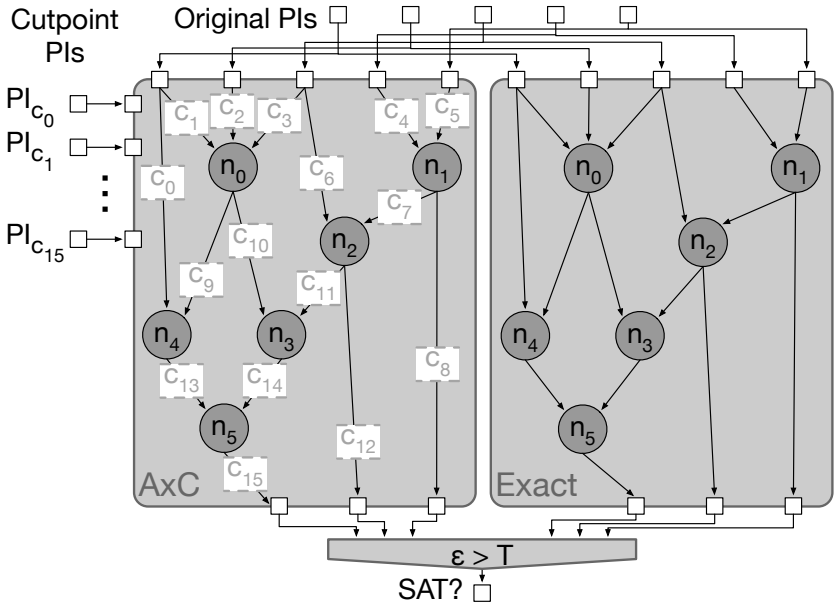


Figure 5.2: Approximation miter with the cutpoints $C = \{c_0, ..., c_{15}\}$ and the corresponding control inputs PI_c. Taken from [106].

5.2.3 *Minimal Unsatisfiable Subsets*

Under the assumption that an activated cutpoint leads to logic removal and thus to savings in the target metric, we can state that the savings will generally increase by activating more cutpoints. Hence, we are interested in finding configurations with as many cutpoints activated as possible while still respecting the quality constraint. That can be cast as a minimal unsatisfiable subset problem: Given an UNSAT problem with a set of constraints Q, $M \subseteq Q$ is a MUS of Q iff M is unsatisfiable and for all $m \in M$ the set $M \setminus \{m\}$ is satisfiable [13]. Note that there can be several MUSes with different cardinality.

Transferred to our ALS setting, we maintain the original primary inputs of the circuit as free variables that a SMT solver can assign and use the additional primary inputs driving the cutpoints as constraints. When setting as constraints $PI_c = 0, \forall c \in C$, the AxC resembles the exact circuit, and the corresponding approximation miter will be unsatisfiable since the AxC is error-free. Now, solving the MUS problem will return a minimal set of constraints M required to keep unsatisfiability. Each constraint $m \in M$ corresponds to a cutpoint that must be deactivated. Vice versa, the set of remaining constraints, which is maximally large, is not required to maintain unsatisfiability; thus, the corresponding cutpoints will be activated to reduce the logic.

In other words, we initially constrain each cutpoint's PI to be deactivated to resemble the exact circuit and construct an UNSAT problem. Then, solving the MUS problem returns a configuration that is guaranteed to be UNSAT and holds a minimal number of cutpoints that must be deactivated to maintain unsatisfiability. Cutpoints that are not part of the returned configuration can be activated safely to remove logic.

For example, assume a design with the cutpoints $C = \{C_0, C_1, C_2\}$ and their corresponding PIs $PI_c, c \in C$. Initially, MUSCAT deactivates all cutpoints by defining the set of constraints $Q = \{PI_{C_0} = 0, PI_{C_1} = 0, PI_{C_2} = 0\}$. Then, after solving the MUS problem, MUSCAT is provided with a minimal set of constraints, for example, the MUS $M = \{PI_{C_0} = 0, PI_{C_2} = 0\}$, that dictates the deactivated cutpoints C_0 and C_2 and allows activating the cutpoint C_1.

5.2.4 *Approximate Logic Synthesis Flow*

Algorithm 5.1 shows the flow for our proposed ALS technique, which takes as inputs the exact circuit O, the used error metric EM, and the error threshold T. First, the algorithm generates the set of cutpoints C and the AxC by augmenting the exact design O with the cutpoints C (Line 2). Next, the approximation miter AM is set up as in Figure 5.2 (Line 3). At this point, the additional primary inputs determining the configuration are actually free variables. In the following step (Line 4), these variables are constrained to $PI_c = 0, \forall c \in C$, which we denote as UNSAT constraints since they guarantee unsatisfiability. Then, based on the approximation miter AM and the UNSAT constraints Q, the MUSes are computed (Line 5). From the constraints in these MUSes, we can directly derive deactivated and, subsequently, activated cutpoints to simplify the corresponding AxCs through logic synthesis (line 6). Finally, Algorithm 5.1 returns the AxC with the highest savings in the target metric.

Algorithm 5.1: Pseudo-code of our ALS technique. Taken from [106].

```
   Input: Exact circuit O, error metric EM, error threshold T
   Output: Best AxC
1  Function muscat( O, EM, T):
2      AxC, C ← addCutpoints( O )
3      AM ← constructApproximationMiter( O, AxC, EM, T )
4      Q ← createUNSATConstraints(C, Q )
5      MUSes ← determineMUSes( AM, Q )
6      AxCs ← generateAxCs( AxC, MUSes )
7      return bestAxC( AxCs )
```

5.2.5 *Discussion on the Insertion of Cutpoints*

There are several related decisions with respect to cutpoint insertion: How many cutpoints should be inserted, and where in the circuit? Which cutpoint design should be used, AND-cut, OR-cut, or AND-OR-cut? Generally, in-

serting cutpoints into every edge of the circuit, as done in the example of Figure 5.2, and using the AND-OR-cut maximizes the potential for savings in the target metric. However, the increased number of constraints might increase the runtime for solving the MUS problem considerably. Even though formal techniques can verify designs with millions of gates and hundreds of inputs in a reasonable time [89, 90], selecting a sufficiently small number of well-positioned cutpoints is desirable to keep the solver runtimes acceptable. Hence, we keep the number of cutpoints sufficiently small while enabling graceful approximations by evaluating a cutpoint's redundancy and immediate savings.

Initially, the input edges of all nodes qualify for cutpoints. In a first step, we evaluate a cutpoint's redundancy based on its appearance in another cutpoint's maximum fanout-free cone (MFFC). Since cutpoints enclosed in the MFFC become obsolete upon activation of the enclosing cutpoint, the enclosed cutpoints pose redundancy and are thus omitted. Furthermore, as all nodes within the MFFC become dangling and thus obsolete upon a cutpoint's activation, the MFFC's size indicates the cutpoint's immediate savings.

Thus, we suggest utilizing the MFFC's size as a ranking criterion for the cutpoints if their number should be reduced further. While other heuristics based on, for example, the fanin or fanout cone's size, also take into account the logic cone affected by constant propagation, the actual effects on the savings are unknown in advance. Furthermore, such heuristics may have a structural bias. For example, cutpoints closer to the circuit's PIs will likely have a larger fanout cone than those closer to the circuit's POs, which introduces a bias towards the cutpoints closer to the PIs. For the fanin cone, this applies vice versa. The MFFC, on the other hand, is more robust against structural bias, and our experimental results show that the MFFC is indeed a reasonable heuristic (see Section 5.3.4).

5.3 EXPERIMENTAL RESULTS

In this section, we present MUSCAT's comprehensive experimental results and analyze how the number of cutpoints, the cutpoint design, and the different heuristics for cutpoint insertion affect the ALS process.

5.3.1 *Implementation and Experimental Setup*

We have implemented Algorithm 5.1 in a fully automated tool flow and made it available as the open-source project MUSCAT. First, we employ Yosys [110] to read the Verilog input design and convert it to a technology-independent netlist. Custom Yosys passes then add the cutpoints and construct the approximation miter automatically. Afterward, we translate the approximation miter into SMT-LIBv2 [11] standard format to ease the addition of constraints

via named assertions[2] and to provide a unified interface to different SMT solvers.

We have experimented with different tools for determining the MUSes, the SMT solver z3 [61] (labeled `z3` in the result plots) that returns the first MUS found and the MUSTOOL [13] that enumerates a set of MUSes within a given time budget (labeled `must_time-budget` in the plots). Internally, the MUSTOOL utilizes z3 as verification engine; the employed algorithm for determining the MUSes, however, differs from z3's procedure.

Table 5.1 lists the used benchmark circuits, and details the number of primary input and primary output bits, and the number of cutpoints available in the design. From all available cutpoints in the gate level netlist, we vary the number of inserted cutpoints from 1% to 100% (suffix *_0.01* or *_1.0* in the plots), using the MFFC's size as a heuristic for the cutpoint insertion if not stated otherwise. The target metric is the circuit area, which ABC [14] determines by technology mapping to the Nangate 45nm Open Cell library. The experiments have been performed on the OCuLUS compute cluster of the Paderborn Center for Parallel Computing (PC²) with Intel® Xeon E5-2670@2.6GHz and 4 Gigabyte main memory.

Table 5.1: Benchmark circuits.

Benchmark Name	Description	#PI / #PO bits	#Cutpoints
rca32	32-bit ripple carry adder	64 / 33	314
absdiff	Absolute difference	16 / 8	203
binsqrd	Square of a binomial	16 / 18	2267
am8	9-bit array multiplier	16 / 16	825
add8u_oFP	8-bit adder from EvoApproxLib [62]	16 / 9	79
mul4	4-bit multiplier	8 / 8	130

MUSCAT offers several parameters that affect the complexity of the problem, the runtime, and the achievable savings. We will present experimental results for different benchmark designs and discuss how the different parameters influence the ALS process in the following. In Section 5.3.2, we scale the number of cutpoints and employ the different tools for solving the MUS problem, z3 and the MUSTOOL. In addition to the number of cutpoints, the MUSTOOL offers the given time budget as an additional degree of freedom for scaling the problem complexity. In Section 5.3.3, we experiment with different cutpoint designs to investigate the impact on the approximate outcome. Section 5.3.4 compares different heuristics for cutpoint insertion to experimentally support our discussion from Section 5.2.1.

5.3.2 *Overall Evaluation*

We experimentally compare our novel approach MUSCAT with AND-cuts against two publicly available state-of-the-art ALS methods: AIG rewriting [22] and the designs in the EvoApproxLib [62]. As the EvoApproxLib

2 We can map named assertions directly to the corresponding cutpoints.

comprises only adders and multipliers of certain bit widths, we only provide a comparison when possible.

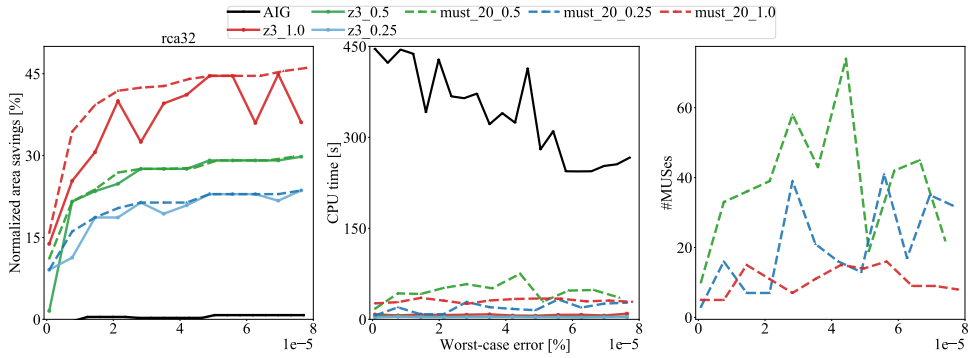
Figure 5.3 shows the experimental results of four benchmark circuits, where the plots on the left side show the area savings normalized to the exact design's area for different WC errors normalized to the circuits' maximum output value. The plots in the center show the runtimes; for MUSCAT, the runtimes reflect all steps of the tool flow, i.e., the cutpoint insertion, the miter generation, the MUS finding, and the synthesis of the AxCs. The plot on the figure's right side depicts the number of MUSes found by `must`, and thus, also the number of generated AxCs.

For the adder `rca32`, AIG rewriting saturates at $\approx$ 1% savings, while our approach achieves significantly higher savings of up to $\approx$ 45%. Even the setups using `z3` that return only a single MUS achieve high savings consistently. If cutpoints are inserted in 100% of the edges, `z3_1.0` shows variations in the savings since only one MUS is selected. Exploring the MUS search space more thoroughly by producing more MUSes with the setup `must` and a time budget of 20s, MUSCAT minimizes the variations in savings and produces very compact AxCs. Exploring the MUS search space and increasing the number of cutpoints is always beneficial for reducing circuit area.

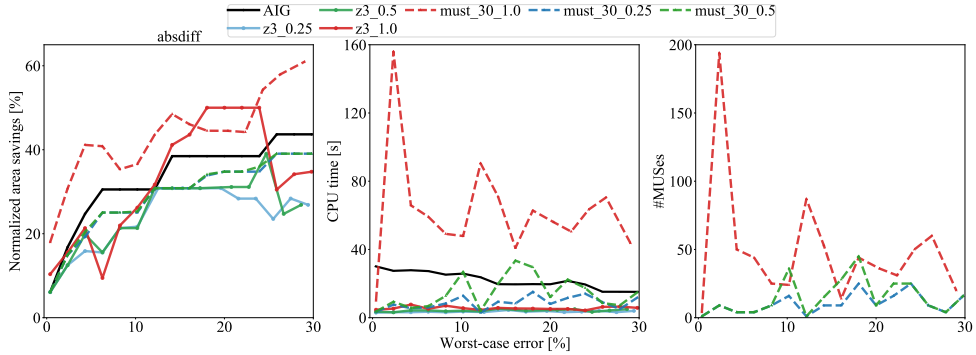
Comparing the ALS runtimes for this benchmark, `z3` averages to 4s, 4.5s, and 7s depending on the percentage of cutpoints, and `must` averages to 18s, 46s, and 30s. Interestingly, for this benchmark, `must` with 50% of the cutpoints endures a longer average runtime than `must` with 100% of the cutpoints. This effect is due to the reduced verification complexity when using 50% of the cutpoints, which, in turn, results in a larger number of MUSes being explored. As a result, MUSCAT generates a larger number of AxCs and the synthesis becomes the dominating factor in MUSCAT. The effect can be observed in the right plot of Figure 5.3a.

AIG rewriting's runtime averages to 343s, and, clearly, our ALS approach is an order of magnitude faster than `AIG` for the benchmark `rca32`. We can further compare the ALS runtimes for `rca32` with published data from other related work. For example, partition & propagate (P&P) [74] reports a runtime of 18s but for determining the nodes' significances only. This figure does not include the rather costly search and simulations for quality verification. Both SALSA [91] and our approach create AxCs valid-by-construction. Unfortunately, SALSA is not open source and the related paper does not provide sufficient details for a re-implementation. However, for the same benchmark, SALSA reported area savings of $\approx$ 15% for an error of $1\% * 10^{-5}$ and a runtime of around four minutes. Hence, for this data point, we outperform SALSA by achieving savings of $\approx$ 25% or $\approx$ 35% within a few seconds using `z3` or `must`, respectively.

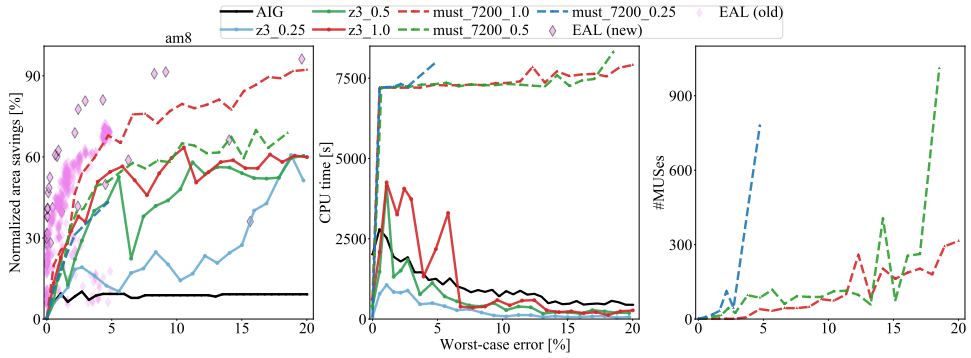
For the benchmark `absdiff`, `must` outperforms `AIG` in terms of area savings, while, in the majority of the cases, `AIG` achieves better results than `z3`. `must_30_1.0` achieves the highest area savings, while `must_30_0.25` and `must_30_0.5` appear to saturate, presumably due to a small number of cut-



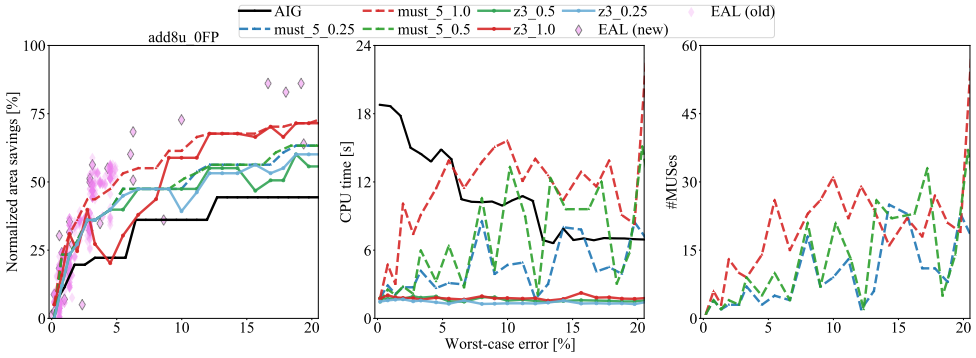
(a) 32-bit ripple-carry adder `rca32`.



(b) Absolute difference `absdiff`.



(c) 8-bit array multiplier `am8`.



(d) 8-bit adder from EvoApproxLib [62] `add8u_0FP`.

Figure 5.3: Comparison of MUSCAT (`z3` and `must`) with different cutpoint percentages to `AIG` and `EAL` for different benchmarks and varying error bounds.

points which, in turn, limits the number of MUSes, as the right plot of Figure 5.3b suggests. `must_30_1.0` determines more MUSes than MUSCAT's remaining setups, and thus, generates more AxCs and shows the highest runtimes, also surpassing `AIG`'s runtimes. In fact, it can be seen that the runtime follows the number of found MUSes, which is expected since more AxCs have to be synthesized. In conclusion, the experiment shows the trade-off between complexity and achievable savings, which allows MUSCAT to adjust to a given time budget.

The results for an 8-bit array multiplier `am8` and the 8-bit adder `add8u_0FP` from the EvoApproxLib are compared against `AIG` and components from the EvoApproxLib (`EAL`). For both benchmarks, `AIG` achieves the lowest savings and our approach achieves up to $\approx 80\%$ higher savings while, in the majority of the cases, enduring lower runtimes when employing `z3`.

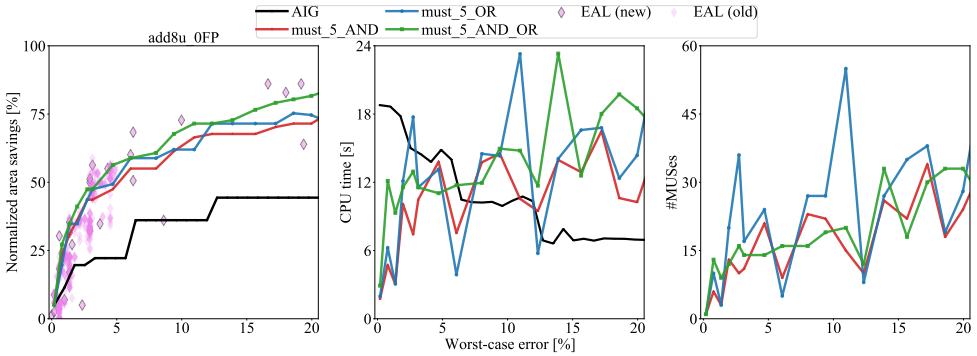
`am8` highlights the benefit of `must` that takes multiple samples from the MUS search space since it achieves higher savings than `z3` – at the costs of higher runtimes. `EAL` achieves better area savings; only for larger error bounds, our method remains competitive. However, `EAL` employs an evolutionary design process over thousands of generations that typically requires very long runtimes but can create efficient designs.

For the benchmark `add8u_0FP`, MUSCAT's area savings are more competitive with `EAL`, and both approaches `z3` and `must` achieve high area savings within a short time. For `must`, it can be seen that the runtimes again follow the number of generated AxCs, which results in some cases in longer runtimes than with `AIG`. Overall, MUSCAT outperforms `AIG` in runtime as well as area savings.

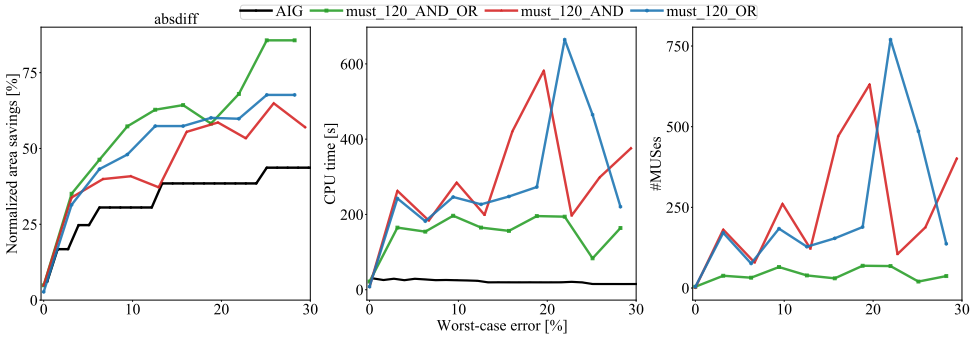
5.3.3 *Evaluation of Cutpoint Designs*

In the next set of experiments, we evaluate different cutpoint designs. Figure 5.4a shows results for approximating the 8-bit adder of the EvoApproxLib `add8u_0FP`. Furthermore, Figure 5.4b shows results for the benchmark `absdiff` for which the MUSTOOL now has a large time budget of 120s to allow for a thorough search space exploration.

For the benchmark `add8u_0FP`, the AND-OR-cut achieves the highest area savings. In fact, MUSCAT even outperforms the EvoApproxLib for some error bounds and states the dominant solution. The results for the benchmark `absdiff` in Figure 5.4b additionally support the insight that the AND-OR-cut is beneficial for achieving higher savings. As discussed in Section 5.2.1, the AND-OR-cut offers greater flexibility as it subsumes the functionality of the AND-cut and the OR-cut. However, at the cost of a higher complexity of the verification problem. The higher complexity is indicated by a smaller number of MUSes found within the same time limit (cf. plots on the right side in Figure 5.4). Nevertheless, even though fewer MUSes, and thus fewer AxCs, have been explored, the increased flexibility appears beneficial, as the larger area savings suggest.



(a) 8-bit adder from EvoApproxLib [62] `add8u_0FP`.

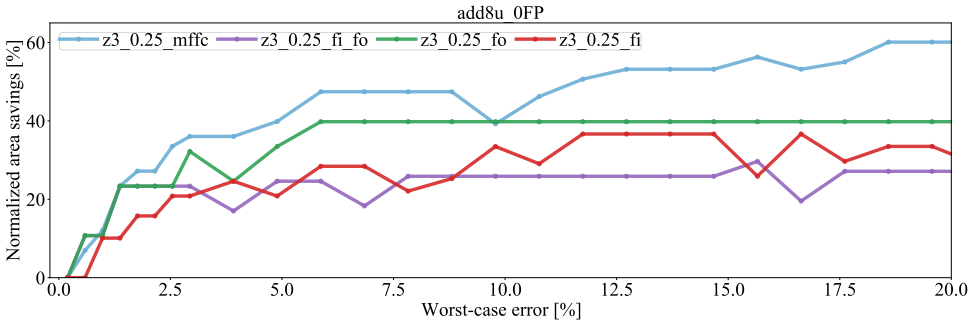


(b) Absolute difference `absdiff`.

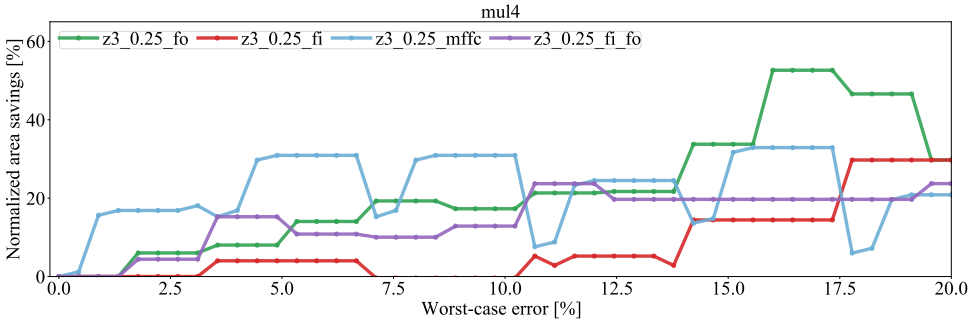
Figure 5.4: Comparison of the method `must` with varying cutpoint designs for 100% of the cutpoints against `AIG` and components of the EvoApproxLib (two versions).

5.3.4 *Evaluation of Heuristics for Cutpoint Insertion*

Figure 5.5 investigates the impact of the cutpoint insertion on the approximate outcome for the benchmarks `add8u_0FP` and `mul4`. We have employed `z3` and ranked all available cutpoints with different heuristics: the cutpoint's size of the MFFC (`mffc`), the fanout cone (`fo`), the fanin cone (`fi`), as well as the aggregate of the fanin and fanout cones (`fi_fo`). From the ranked cutpoints, we then only inserted the subset of the 25%-best cutpoints. The results for `add8u_0FP` show significant differences in the achieved savings for the heuristics. The heuristic `mffc` performs best in the experiment by consistently achieving high savings with slight deviations for small or large worst-case errors, i.e., the heuristic enables graceful approximations even though MUSCAT uses a small number of cutpoints. The heuristic `fo` achieves high savings for small WC errors; however, the heuristic appears to saturate at savings around 40% for larger errors. While showing the most considerable deviations, the heuristic `fi` improves towards larger error bounds on average. However, the achieved savings are significantly smaller than those achieved with `mffc`. Overall, the heuristic `fi_fo` performs worst in the experiment.



(a) 8-bit adder from EvoApproxLib [62] `add8u_0FP`.



(b) 4-bit multiplier `mul4`.

Figure 5.5: Comparison of different heuristics for the cutpoint insertion.

For the 4-bit multiplier `mul4`, the heuristic `mffc` performs best up to a WC error of 10%. For larger error bounds, `mffc`'s performance stagnates or even declines, and the heuristic `fo` achieves the highest savings. Both heuristics `mffc` and `fo` outperform on average the heuristics `fi` and `fi_fo`. Overall, the `mffc` achieves good results, and thus, justifies cutpoint ranking by the MFFC's size if only a subset of all available cutpoints should be inserted; `fo`, however, represents a viable alternative.

5.4 CASE STUDY: SQUARE OF A BINOMIAL

In this section, we conduct a case study on a complex design for formal verification methods. We analyze the impact of the SMT solver on the outcome and elevate MUSCAT to the register-transfer level to abstract the complexity of the design through which we anticipate reductions in runtime.

5.4.1 *Overview*

The most challenging design in our benchmark set (see Table 5.1) is `binsqrd` that contains two adders and three multipliers, which are known to be hard to verify [90]. For this benchmark, we conduct a case study to motivate future directions for MUSCAT to enable the approximation of complex designs in shorter time at the cost of potentially lower approximation granularity. We

use AND-cuts, experiment with raising the design's level of abstraction, and compare the performance of different SMT solvers.

A SMT solver essentially combines a SAT solver with one or more theory solvers, enabling the SMT solver to operate at a higher level of abstraction [12] than a SAT solver that operates on Boolean variables and operations. In fact, SMT solvers verify the satisfiability of a formula that may contain operations from various theories, e.g., bit vectors or integer arithmetic, and use the highly-specialized theory solvers for efficient solving (cf. Section 2.2). We exploit the expressive power of modern SMT solvers and elevate MUSCAT to the register-transfer level, which brings two advantages.

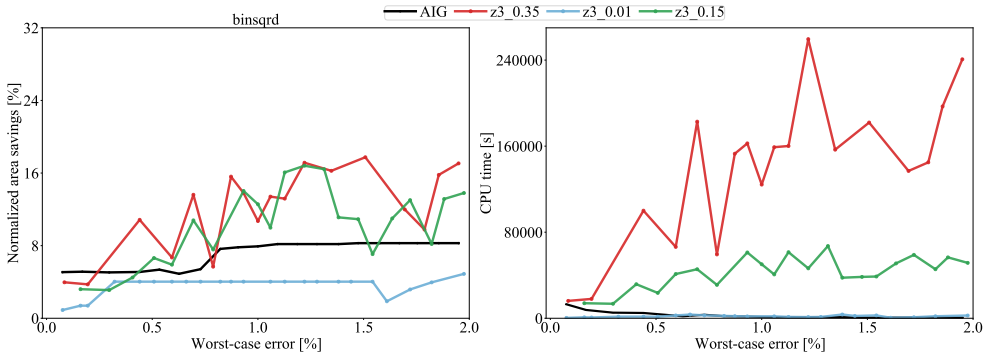
Firstly, the employed SMT solver can exploit its available potential through the ability to handle different theories, e.g., arithmetics or bit vectors, to solve a generalized SAT problem. In this way, the benchmark's behavioral description of the arithmetic operations can be preserved at register-transfer level (RTL), rather than describing the operations in Boolean logic at gate level, which results in a complex verification problem as the runtimes in Figure 5.6a show. As a result, the solver can handle the arithmetics more efficiently by utilizing the highly-specialized theory solvers for such predicates and their conjunctions.

Secondly, the overall complexity of the problem is abstracted since the design contains fewer details, i.e., fewer components and connections, as it is not described via Boolean logic over individual bits, which is reflected in the reduced number of available cutpoints of 115 instead of 2267 as in the benchmark's gate level netlist. However, the cost for the higher abstraction is that the cutpoints affect larger portions of the design instead of individual logic gates, which may lead to less graceful but coarse-grained approximations.

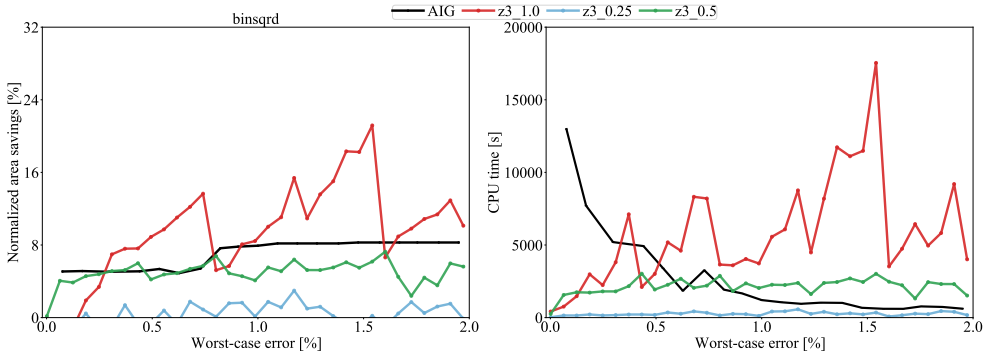
5.4.2 *Experimental Evaluation*

Figure 5.6a shows the area savings and the runtimes for MUSCAT as presented in this chapter, i.e., MUSCAT operates on the gate level netlist. Due to the complexity of the benchmark, we only generate a single AxC with `z3` and consider a smaller percentage of inserted cutpoints. In most cases, MUSCAT achieves better area saving than AIG rewriting. However, it can also be seen that limiting the number of cutpoints too aggressively or extracting only one sample from the MUS search space may lead to suboptimal results, e.g., `z3_0.01` saturates at low savings or the achieved savings of a setup deviate largely. As the runtime plot shows, AIG rewriting has a significantly shorter runtime than `z3_0.35` and `z3_0.15`; the runtime of `z3_0.01` is comparable to AIG rewriting's runtime. Overall, the runtimes average to 1755s, 43146s, and 139965s for the different cutpoint percentages, and AIG rewriting has an average runtime of 1327s.

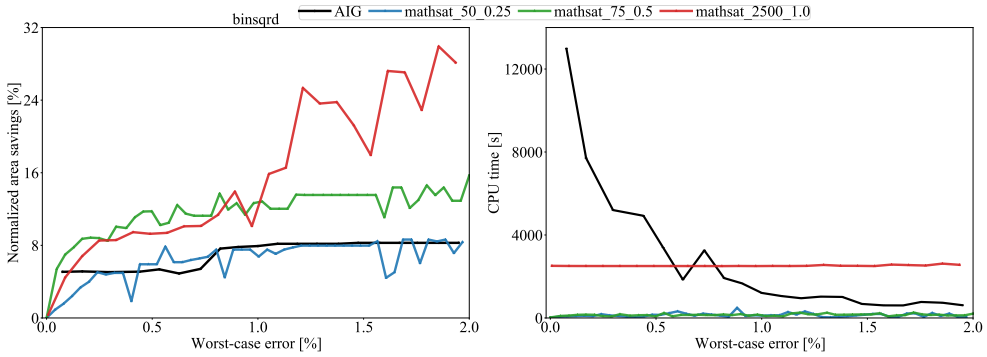
Figure 5.6b shows the results for MUSCAT at RTL with `z3` determining a single MUS. `z3_1.0` achieves the highest area savings, which are comparable to MUSCAT's gate level implementation. However, the savings deviate largely. The area savings for `z3_0.50` and `z3_0.25` are more steady but lower. While



(a) Approximation of the gate level netlist.



(b) Approximation of the RTL description with z3.



(c) Approximation of the RTL description with mathsat.

Figure 5.6: Experimental results for the benchmark `binsqrd`.

`z3_0.50` achieves similar savings to AIG rewriting, `z3_0.25` achieves the lowest savings. In fact, the area curves suggest a recurring saw tooth pattern, which we attribute to internal heuristics used in the SMT solver. Further research into the heuristics may reduce the deviations and lead to more steady savings over the increasing WC error.

MUSCAT's runtimes are significantly reduced by one order of magnitude, and average to 261s, 2147s, and 5777s for 25%, 50%, and 100% of the cutpoints, respectively, resulting in a $\approx 24\times$ speed-up for 100% of the cutpoints. Thus, the experimental results confirm our intuition of lower runtimes at the costs of lower approximation granularity.

In order to evaluate the impact of the formal verification engine on the performance, we have conducted experiments with the SMT solver MathSAT [26]. By default, MathSAT provides no support for MUSes [39]; thus, we integrated the solver into the MUSTOOL as the underlying verification engine. Figure 5.6c shows the results achieved with MUSCAT and MathSAT. Remarkably, a time budget of only 50s or 75s is sufficient for MathSAT with 25% or 50% of the cutpoints, respectively, to determine multiple MUSes from which MUSCAT generates compact AxCs. When using 100% of the cutpoints, a substantially higher time budget of 2500s was required to generate compact AxCs consistently; for lower time budgets, some of the experiments did not complete and MUSCAT generated no AxCs. The runtimes average to 137s, 147s, and 2519s for 25%, 50%, and 100% of the cutpoints, respectively.

Interestingly, employing a smaller number of cutpoints is beneficial for the approximate outcome for smaller error bounds since better area savings are achieved in a significantly shorter time. For larger error bounds, however, the setup using 100% of the cutpoints achieves the highest area savings among all experiments – at the cost of higher runtimes. Compared to AIG rewriting, MUSCAT with 75% of the cutpoints consistently achieves larger area savings at lower runtimes.

We also performed experiments with the MUSTOOL and z3 as underlying verification engine and provided the same time budget as for MathSAT. However, in this setup, no MUSes have been found, and thus, no AxCs have been generated. Thus, for this benchmark, MUSCAT combined with MathSAT outperforms the other approaches by several orders of magnitude in runtime while achieving higher area savings. The experiment highlights the impact of the underlying formal verification engine on the approximate outcome and motivates further research into selecting the verification engine most suitable for the input design.

In conclusion, the case study on the benchmark circuit `binsqrd` demonstrates that MUSCAT scales to complex designs by raising abstraction to the register-transfer level to reduce approximation granularity and leverage the potential of SMT solvers. As a result, MUSCAT reduces the runtimes significantly. The main factors for reducing the runtimes are 1) the simplified verification problem through a reduced approximation granularity and 2) the SMT solvers that can leverage the theory solvers more efficiently at RTL. However, as the case study comprises only a single benchmark circuit, fur-

ther experiments with different benchmarks have to be conducted to confirm the results.

5.5 CONCLUSION

In this chapter, we have presented the approximate logic synthesis method MUSCAT to generate valid-by-construction approximate circuits. Our approach augments an input design by cutpoints that, when activated, allow for logic optimization through dangling node removal and constant propagation. Formulating the AxC's construction as an unsatisfiability problem, we exploit the performance of modern formal verification engines to determine minimal unsatisfiable subsets that translate to sets of cutpoints that should be activated.

In our experiments, MUSCAT achieves up to 80% higher savings than AIG rewriting while enduring lower runtimes in most experiments, and remains competitive with a computationally expensive evolutionary algorithm, the EvoApproxLib. In fact, MUSCAT allows for scaling the runtimes by either scaling the number of employed cutpoints or exploring the MUS space more or less thoroughly. We have also compared MUSCAT to SALSA on a design point taken from literature. This comparison indicates that MUSCAT can achieve higher area savings at substantially lower runtimes. Finally, we conducted a case study on a complex benchmark design. We elevated MUSCAT to the register-transfer level and evaluated the impact of the formal verification engine on the approximate outcome. In this way, we achieved significant reductions in runtime by several orders of magnitude and exemplified MUSCAT's potential for approximating more complex designs on a higher level of abstraction.

6

PROOF-CARRYING APPROXIMATE CIRCUITS

6.1 OVERVIEW

When performing approximate logic synthesis (ALS) for generating an approximate circuit (AxC), an important problem is determining the actual quality of the resulting circuit to analyze the suitability of the AxC in a specific application context. For some applications, the specified constraints are soft, for example, when statistical constraints such as the mean squared error are specified (cf. Section 2.2). However, adhering to the quality constraints is crucial for other applications, and these applications thus require a guarantee on the constraints. These scenarios rule out testing-based approaches since exhaustive testing of all possible input combinations, e.g., when specifying a worst-case (WC) error bound, is clearly infeasible. Instead, formal verification methods must be employed to guarantee the specified quality constraints of the AxC. Unfortunately, most of the existing automated ALS frameworks use a testing-based approach [48, 50, 65, 94], and only a few ALS frameworks generate AxCs with guaranteed error bounds [22, 69, 91]. While being conceptually much stronger than testing-based approaches, formal verification techniques also tend to lead to considerable runtimes.

We expect in the near future that approximate components will be offered and traded as intellectual property (IP) cores, possibly in libraries, as it is the case for other digital circuits. Today such approximate component libraries already exist for small arithmetic components [19, 62, 70, 81], but the concept can easily be extended in scope. In this scenario, a pressing question for the party that purchases an approximate IP core and integrates it into a larger design is whether the core actually meets the specified quality constraints. Even though some of the existing ALS frameworks provide a guarantee on the quality through formal verification, the consumer has no proof that the guarantee actually applies to the purchased core, i.e., the consumer of the IP core has to trust the producer and the producer's tool flows. Naturally, trust in a producer or producer's tool flows is also an issue for core characteristics such as hardware area, energy consumption, and delay. Accuracy, however, is a functional parameter and thus more critical, as a too low accuracy can render a design completely useless for a specific application.

As a novel contribution, we present the concept of *proof-carrying approximate circuits* and propose to apply the technique of proof-carrying hardware (PCH) [30] to approximate circuits. The producer annotates the AxC with

a formal proof of its quality and offers the result as IP core, ranging from simple arithmetic components to complex applications. The purchasing party can verify the proof and, if successful, be certain about the core's quality. A key aspect of PCH is that the effort for verifying the proof is much lower than for creating the proof, shifting the burden of the formal verification task to the producer. As a result, no trust in the producer is needed and the consumer can verify the core's quality at a fraction of the computational costs.

This chapter makes the case for proof-carrying approximate circuits and elaborates on this concept, including the required tool flows for the producer (vendor) and consumer (purchasing party) of an approximate circuit. We base the elaboration on an early adoption of our CIRCA framework presented in Chapter 3 that resulted from a Master's thesis [100], which we configure for the approximation of sequential circuits. We then guarantee the error bounds via inductive verification and demonstrate the proof-carrying approximate circuits scenario.

We have presented proof-carrying approximate circuit first in our workshop paper [104] and eventually published a journal version [105]; this chapter is based on these publications, and cites and extends their discussions. My contribution to this work is the approximate computing part, while my colleague, Tobias Wiersema, contributed the proof-carrying hardware side of the project, which he has further detailed in his dissertation [97].

6.2 PROOF-CARRYING HARDWARE

Proof-carrying hardware was first introduced by Drzevitzky et al. [30], who described its context as the interaction of two parties, a hardware module (or IP core) producer and a consumer, as depicted in Figure 6.1. The consumer requests a hardware module with a function according to a given design specification and for which certain constraints must hold. These constraints are also denoted as (safety) properties. Instead of forcing the consumer to trust the received module – or to formally verify that the property actually holds at their own expense – the PCH technique requires the producer to attach evidence of a formal proof (often denoted as *proof certificate*) to the IP core. The consumer then can retrace the proof steps at a fraction of the regular computational verification cost since making a formal proof certificate is (usually) much harder than verifying a given proof certificate. Thus, the cost of trust shifts to the producer who typically has sufficient computational resources and formal verification methods already in place.

Note that the application of the PCH method actually evaluates the trustworthiness of all gray components in Figure 6.1, so it even works if the consumer trusts neither the producer, the module, the proof, nor the communication channel via which they receive both. Here, the key element is that by specifying the design and the property, the consumer forms a proof base against which the received proof certificate can be verified. Any tampering done in any of the untrusted gray areas in Figure 6.1 will be detected by PCH

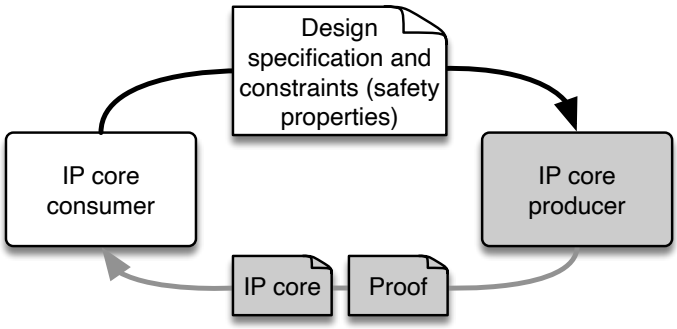


Figure 6.1: Proof-carrying hardware flow.

because, in such a case, either the module or the proof will not match the consumer's proof base, or the proof will be corrupt and cannot be validated. A successful validation using the PCH technique thus guarantees ultimate trustworthiness of the transmitted module with respect to the agreed-upon property.

Since its inception, the PCH concept has been applied to several types of digital circuits at different abstraction levels. For example, Drzevitzky et al. [30] focused on combinational circuits and applied a satisfiability (SAT)-based tool flow with proofs encoded as resolution proof traces. Later on, Drzevitzky [29] extended their approach to bounded sequential verification using a time frame expansion technique. Finally, Isenberg et al. [43] showed a PCH flow with induction as proof principle, where inductive invariants serve as evidence of the formal verification. This PCH flow is not limited by time frame expansion and can thus provide stronger, i.e., for an unbounded number of cycles, guarantees for sequential circuits. Also, the shift of workload to the producer was investigated by Isenberg et al. [43]. For example, in the most pronounced case, the producer required 57 minutes to generate the proof, whereas the consumer only needed 1.4 seconds to verify its validity. In other words, instead of spending 57 minutes to perform a full verification, the PCH scheme allowed the consumer to reduce their workload by 99.95% down to 1.4 seconds *without loss of verification strength*. While all these approaches applied PCH at the level of an FPGA's configuration bitstream, i.e., essentially a placed and routed netlist, Love et al. [51] have created a concept to create manual proofs for a subset of the Verilog HDL, bringing PCH to the source code level. Since after a successful verification the producer has to run various design automation tools and transmit the resulting module to the consumer, the trustworthiness of the latter two steps can not be guaranteed. Actually, Krieg et al. [46] showed the vulnerability of source code level PCH by describing an attack that inserts trojans into hardware modules although they have been successfully verified at the source code level. Ahmed et al. [3] showed that PCH at the configuration bitstream level can indeed detect such trojans.

The PCH concept requires both the consumer and producer to agree on formalisms to specify the circuit function and the property to be proven. It is important to note that the PCH concept is not restricted to a specific property.

While most PCH works provided proofs for the property equivalence of a combinational or sequential circuit, respectively, to its specification, Wiersema and Platzner [98] demonstrated the verification of the non-functional property of meeting pre-defined response time limits for sequential circuits. In the context of approximate computing, we seek to guarantee that an AxC does not violate a pre-defined bound on some error metric by relaxing the *equivalence* property to *equivalence up to some bound* [89] (cf. Section 2.2) .

6.3 PROOF-CARRYING APPROXIMATE CIRCUITS

We propose to apply the PCH concept to approximate circuits to formally guarantee error bounds and allow any recipient of such a circuit to confirm its trustworthiness without needing to trust the producer at a fraction of the cost of a full formal verification. The left side of Figure 6.2 shows the general form of interaction between a producer and a consumer for such proof-carrying AxCs. We envision two different scenarios.

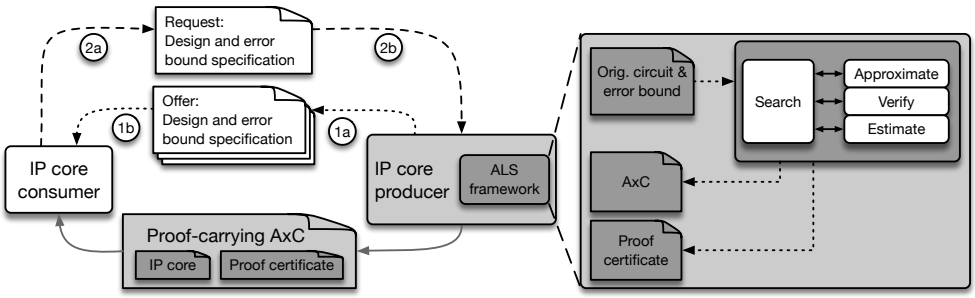


Figure 6.2: The left side shows the two scenarios, ① and ②, for proof-carrying approximate circuits. The right side of the figure shows the conceptual flow of an approximate logic synthesis framework utilized by the producer. Adapted from [105].

Scenario ① is described by dotted and solid lines on the left side in Figure 6.2. The producer generates AxCs with certain error bounds and offers these as IP cores, e.g., as approximate function libraries. Additionally, the producer prepares proof certificates that the AxCs actually meet the targeted error bounds. To enable potential users (consumers) to find such approximated IP cores, the producer needs to publish both the design specification and a specification of the error bound. When the consumer has decided on such an approximated IP core, the producer packages the requested IP core with the proof certificate and sends it off to the consumer. Upon successfully verifying the proof certificate, the consumer holds a guarantee for the core's quality. For PCH schemes, these certificates are always generated in a way that allows for a much faster validation than generation. For example, some PCH works employ resolution proofs for the satisfiability of a Boolean formula in conjunctive normal form as the certificate. To check whether the given sequence of resolution steps in such a proof actually leads to the empty clause requires significantly less computational effort than generating that sequence in the first place. This asymmetry in computational complexity

between generation and validation is the core of the workload shift between consumer and producer, typically leaving the consumer with a validation problem that is significantly easier to solve than a full formal verification of the desired circuit properties.

The approximate component libraries published in [19, 62] state examples for scenario ①. These libraries contain approximate adders and multipliers, which provide different characteristics in terms of hardware area, delay, power consumption, and quality. Furthermore, the authors note that each component has been verified using a SAT solver to guarantee the circuit's quality. A potential consumer of these components, however, still has to trust the producer and the given specifications of the component. In our proof-carrying AxC scenario, the bundle of approximate component and proof certificate allows the consumer to verify the specifications quickly, and thus, gain an ultimate level of trust. There are also examples for libraries that do not give any guarantee, such as the adders and multipliers published in [81] and [70]. These could also be used for scenario ① by translating the contained quality characterization into suitable error bounds. Note that the components offered in an approximate component library for scenario ① are theoretically not limited in size or scope and can range from simple arithmetic units, e.g., adders and multipliers, to more complex IP cores, e.g., discrete cosine transform cores [84].

Scenario ② is indicated by dashed and solid lines on the left side in Figure 6.2 and is more dynamic as it lets the consumer request the creation of an approximated IP core with a specific error bound. To enable potential producers to agree to such contract work, the consumer has to publish both the design specification and a specification of the error bound. Next, the contracted producer creates the approximated IP core, which again can be a simple combinational circuit or a complex sequential design, along with the proof certificate, and sends it off to the consumer. Additionally, the consumer might want to set constraints on parameters such as area, delay, or energy consumption that should be achieved by tolerating the specified error. This process is, however, not covered by Figure 6.2.

Proof-carrying approximate circuit flows can be used for different target technologies at the consumer's side. If the consumer develops in custom or an ASIC design style, the runtimes required for the verification steps will possibly not negatively affect overall design time. If, on the other hand, the consumer utilizes reconfigurable accelerator technology, time scales could be dramatically shortened, requiring efficient tool flows. The extreme case *on-the-fly computing* [40] performs all steps of scenario ② at runtime.

A precondition for both scenarios is that the design and the error bound are formally specified. In both scenarios, the consumer does not need to trust the producer, the producer's techniques and tool flows, or the transmission of the proof-carrying approximate circuits (depicted in gray in Figure 6.2). Due to the PCH approach, the consumer side will detect any tampering with the circuit or the proof. Even matching modifications of the circuit and the proof will be detected if they guarantee a property different from the specified one.

In order to successfully implement the proof-carrying approximate circuit concept, several requirements have to be met:

1. The employed approximation techniques have to generate circuits for which definable quality constraints, i.e., error bounds, can be formally guaranteed.
2. These error bound guarantees must be transformable into proof certificates, which can be transmitted with the circuit.
3. The verification of the proof certificates (the consumer's workload) should be faster than formally verifying the circuit's error bound in the first place (the producer's workload), enabling the core benefit of our approach for the consumer: gaining trust in AxCs at a fraction of the verification costs.

6.4 VERIFICATION-BASED APPROXIMATE LOGIC SYNTHESIS

In this section, we will first elaborate on the verification-based ALS process. Then, we will discuss the proof certificates in more detail.

6.4.1 *Approximate Logic Synthesis and Quality Assurance*

The right-hand side of Figure 6.2 depicts the general flow of a verification-based ALS framework. The ALS process starts with an original (unapproximated) circuit and an error bound as inputs, and performs, for example, search-based ALS as described in Section 2.1 or Section 3.1. We leverage the CIRCA framework (see Chapter 3) for experimental validation. Conceptually, however, our approach can utilize any ALS process that applies functional approximation and employs a formal verification method which provides a proof certificate to fulfill the PCH requirements.

Approximations often target arithmetic components in the data path of an application. Concepts for constructing approximate arithmetic units and approximate component libraries comprising adders and multipliers have been presented in the past of which some even provide guarantees on the quality constraints [4, 19, 62, 70, 81, 116]. However, two issues remain when using such components in a larger circuit: 1) The user has to either trust the given guarantee or run a full verification process to ensure the quality. 2) The errors of the individual components propagate through the circuit, possibly amplifying or canceling out, which means individually verifying the quality of candidates in the circuit is not sufficient; instead, the overall circuit has to be verified (see Section 2.2). To assure the quality of an AxC, i.e., to verify whether it satisfies the error bound, we form a sequential quality constraint circuit (SQCC), as described in Section 3.4.5, and generate a proof certificate for the AxC as evidence of a successful formal verification.

Evaluations of AxCs that rely on testing generally utilize statistical error metrics, such as the mean absolute error, the mean relative error, or the error rate. In formal verification, however, statistical metrics are extremely

hard to guarantee because the SQCC needs to be extended to count error information over all input vectors (cf. Section 2.2). In fact, statistical error metrics do not state a SAT problem but a much more complex #SAT (counting SAT) problem [90], of which current #SAT solvers can only process small instances [89]. Vašíček [90] discusses algorithms for determining statistical error metrics for combinational circuits using formal methods and highlights the complexity of determining such error metrics with experiments on small-scale circuits. For sequential circuits, the complexity may even increase to a multitude of #SAT problems, which, to the best of our knowledge, currently renders the approach infeasible for real-world circuits. Interestingly, Chandrasekharan et al. [23] use an approximation miter similar to ours and try to guarantee the mean absolute error; their initial experiments, however, did not conclude for practical designs.

Our research focuses on a non-statistical error metric of modest complexity, the WC error – one of the most commonly used error metric in approximate computing [111]. Conceptually, however, our approach is not limited to a specific error metric; in fact, the scalability of our approach of proof-carrying approximate circuit depends on 1) the ALS method and 2) the employed verification engine. 1) dictates the number of verifications during ALS, which directly impacts the overall runtimes [101, 102]. Furthermore, the runtime of a single verification in the approximation process depends on 2), and, depending on the verification problem, the differences between the methods can be significant [15]. In summary, our approach inherits the scalability of the underlying ALS and verification engine but does not add any bottleneck.

6.4.2 *Creating Proof Certificates*

Based on the SQCC, the producer must transform the error bound guarantee into a proof certificate that is sent to the consumer, i.e., construct the proof via a full formal verification. The proof certificate has to enable a consumer to objectively draw the same conclusion as the original verification, only much faster. Since the producer is inherently untrusted in this method, the consumer has to construct the proof base themselves and independently from the producer by combining their own design and error bound specifications with the AxC's implementation extracted from the IP core's netlist into an SQCC. The consumer can then take their SQCC and the received certificate and check that 1) the proof bases match and 2) the proof is correct. Note that both parties have to agree on a common representation and transformation rule set to establish a working process, e.g., by agreeing on a set of tools to use and allowed sequences of structural optimizations that the producer can employ to make challenging verification problems tractable.

As the SQCC for combinational circuits is also combinational, it can be directly verified by proving unsatisfiability of the SQCC. The proof base, which both producer and consumer determine independently, is the set of input clauses in the SQCC's conjunctive normal form (CNF) representation. The certificate is the trace of all clausal resolutions that imply the empty

clause from the CNF formula. The consumer thus needs to transform their own SQCC into CNF and apply the clausal resolutions step-by-step as listed in the certificate. Finally, the error bound is verified if the empty clause is obtained.

For sequential circuits, the SQCC has to be verified using sequential verification methods, which prove that the error bound holds in all reachable states of the SQCC. Since the set of reachable states can be huge, an enumeration of these states and explicit checks for the error bound is infeasible. Hence, we employ the efficient property-directed reachability (PDR) [31] implementation from ABC [14]. PDR is an inductive solving technique that first checks whether the property to be proven holds in the initial state of the circuit (induction anchor, initiation) and second whether the property unconditionally holds for all states after a transition, provided it held in the previous states (induction step, consecution). However, since properties of interest, such as guaranteed error bounds, are not necessarily inductive, PDR searches for a so-called *inductive strengthening* of the original property. In particular, the circuit could have states – not reachable from the initial state – for which the error bound holds, but not for their successor states. In such cases, the consecution step of the induction would fail. The inductive strengthening of the original property is a more restrictive property that holds in a subset of the states in which the original property holds and, additionally, is inductive. The resulting strengthening is also called an *inductive invariant*. Technically, PDR breaks the induction on the circuit state down into a multitude of SAT problems, which, if the SQCC is unsatisfiable, will yield such an inductive invariant of the original property. Then, ABC outputs the inductive invariant, which we use as certificate.

The consumer then builds the proof base by constructing the SQCC, and if the producer has performed allowed structural pre-processing on the circuit, the consumer has to apply the same sequence on their version. In our implementation, we allow the producer to apply structural optimizations in their effort to minimize the consumer's workload through their pre-verification, ranging from minor cone-of-influence reductions up to the more heavy-duty pre-processing of ABC's *dprove* command. In case these steps alone solve the SQCC, the producer cannot and does not need to generate a certificate, but the consumer also has to perform no more than the previously agreed-upon structural circuit optimizations to prove the AxC quality themselves. Otherwise, the consumer checks three conditions, namely that the received invariant 1) holds in the initial state, 2) is inductive (i.e., the induction step holds), and 3) is indeed a strengthening of the SQCC's derived CNF formula. All three checks translate into quite simple SAT problems, which the consumer can solve much faster than the multitude of SAT problems the producer had to solve to arrive at the invariant. For a more in-depth description of the technique, we refer to [43].

6.5 EXPERIMENTAL RESULTS

We validated the proof-carrying AxC approach through experiments, using AxCs generated by the CIRCA framework with the benchmark circuits listed in Table 6.1. The candidates have been manually annotated to specify subcircuits subjected to approximation. In addition, CIRCA has been configured to use hill climbing as search method and precision scaling or AIG rewriting [22] as approximation techniques to prove the general applicability of our approach. The circuit area was the minimization objective, and the WC error was the error metric, varying from 0.025% to 2.0% of the circuit's maximal possible output value. Since the work of proof-carrying AxCs focuses on verifying AxCs with PCH, and thus, the achieved savings in the ALS process are secondary, we only summarize the ALS results: Overall, CIRCA achieved area savings of up to $\approx$ 26% through approximation while guaranteeing the specified error bound. The average runtimes of the entire ALS process ranged from around one minute up to five days.

As described in Section 6.4.2, we employ ABC's verification techniques, which, in recent years, dominated the `single property` track of the Hardware Model Checking Competition [15], where verification problems from industry had to be solved, proving ABC's performance and scalability, and thus, the scalability of our approach to industrial-strength verification problems. Our verification experiments have been repeated ten times on the OCuLUS cluster of the Paderborn Center for Parallel Computing (PC²) with a time limit of seven days for each run of the flow. The cluster runs Scientific Linux 7.2 (Nitrogen), provides 16 Gigabyte main memory per job, and comprises nodes with an Intel® Xeon E5-2670@2.6GHz. Table 6.1 lists for all benchmarks, approximation techniques, as well as error bounds the runtimes of the producer flow, the consumer flow, and the reduction of computation time the consumer experiences over full verification.

Our experimental results show the significant impact our approach of proof-carrying AxCs has on the consumer's runtimes. The observed runtime reductions for the consumer highlight the effectiveness of our approach and range from 0.00% (structural optimization solved the SQCC) to 99.83%, averaging to around 71.03%. While the producer endures the runtimes of the full formal verification, the consumer benefits from the workload shift and is able to quickly verify the AxC's quality with the proof certificate.

As the different approximation techniques modify the structure of the circuit differently, and thus the complexity of the verification problem, significant differences in runtimes of the verification processes among the same benchmark can be observed, cf. `fir_gen`, 0.25%. In fact, our results show that the runtimes of the verifications are neither strictly increasing nor decreasing with increasing error thresholds, cf. `ternary_sum_nine` and `butterfly`.

The 4-tap FIR filter benchmark `fir_gen` implements four generic 9-bit multipliers, which are known to state a difficult verification problem [90]. With increasing error bounds, however, the verification problem seems to become less complex. On the other hand, the 16-tap FIR filter `fir_pipe_16`,

Table 6.1: Experimental results for the proof-carrying AxC approach. Taken from [105].

		AIG rewriting			Precision Scaling		
Circuit	Error	Runtime [s]		Reduction	Runtime [s]		Reduction
name	bound [%]	Consumer	Producer	[%]	Consumer	Producer	[%]
butterfly	0.025	1.58	3.35	52.44	1.64	5.33	51.59
	0.25	1.58	7.23	78.13	1.49	3.44	37.99
	0.50	1.55	4.60	66.23	1.56	4.77	64.01
	1.00	1.52	4.53	66.47	1.46	2.99	49.20
	1.50	1.53	4.07	60.75	1.47	2.88	45.12
	2.00	1.51	4.22	64.14	1.44	2.35	38.27
fir_gen	0.025	2.20	7.20	69.36	1.67	4.21	60.21
	0.25	36.41	193.84	79.55	1.33	2.55	46.58
	0.50	15.03	65.67	77.09	1.31	2.19	40.03
	1.00	9.89	44.43	77.71	1.25	2.16	42.53
	1.50	6.30	38.06	82.72	1.26	1.79	29.71
	2.00	4.18	15.68	73.31	1.11	2.16	46.03
fir_pipe_16	0.025	2.43	2.68	7.71	2.68	2.68	0.22
	0.25	4.69	322.85	98.54	37.53	219.95	82.92
	0.50	23.76	2245.49	98.93	80.34	1982.81	94.73
	1.00	145.18	1489.96	90.24	85.86	2132.54	93.19
	1.50	134.46	2215.58	93.44	97.71	2415.09	94.76
	2.00	215.10	2607.77	91.47	81.86	2356.28	96.29
pipeline_ add	0.025	0.08	0.08	0.00	0.11	0.35	68.05
	0.25	0.13	1.34	84.11	0.14	0.45	70.10
	0.50	0.13	1.41	84.13	0.12	0.37	66.80
	1.00	0.13	1.16	76.02	0.13	0.37	66.33
	1.50	0.13	1.33	76.66	0.12	0.39	69.45
	2.00	0.15	1.26	80.57	0.12	0.30	62.18
rgb2ycbcr	0.25	1.95	23.93	91.84	2.43	7.09	65.60
	0.50	1.94	27.98	93.05	2.35	5.95	60.47
	1.00	1.82	18.49	90.13	2.43	6.21	60.72
	1.50	1.87	21.37	91.24	2.33	5.33	56.26
	2.00	2.10	24.43	91.38	2.28	5.16	55.73
ternary_ sum_nine	0.025	0.48	0.49	1.41	0.61	0.77	21.54
	0.25	59.42	506.69	87.85	0.48	6.43	38.96
	0.50	6.70	709.78	99.05	0.19	3.65	90.01
	1.00	4.01	2505.66	99.83	0.35	10.17	71.03
	1.50	0.79	251.26	99.68	0.23	4.97	95.49
	2.00	2.19	154.41	98.58	0.18	2.10	91.28
weight_ calculator	0.25	1.57	1.57	0.00	16.97	1640.97	98.81
	0.50	9.71	541.17	97.90	18.06	1489.40	98.82
	1.00	32.44	1399.23	97.60	13.21	1153.46	98.84
	1.50	15.38	945.05	97.68	9.76	819.14	98.61
	2.00	40.93	2084.28	97.92	10.02	1377.77	99.25

which implements constant multipliers, shows the opposite behavior. Here, the verification complexity increases with the error, and thus, the verification runtimes. In both cases, however, our approach reduces the consumer runtimes.

The benchmarks with pipelines (`fir_pipe_16` and `weight_calculator`) highlight the correspondence of the AxC verification challenge to the amount of changes in the circuit. For small error thresholds, CIRCA could only modify the last-most pipeline stages due to error propagation effects in the pipeline, which results in significantly easier verifications than larger error bounds, which lead to changes also in earlier pipeline stages, with discontinuities presumably due to the complex interaction of the stages.

Compared to the runtimes of the entire ALS process, where a large number of AxC is being generated and verified, the runtime of a single formal verification on the producer's side is comparably small. Hence, as discussed in Section 6.4.1, our approach of proof-carrying AxCs is not the runtime-limiting factor in the flow.

6.6 CONCLUSION

In this chapter, we have made the case for proof-carrying approximate circuits. Using this novel concept, producers of approximated IP cores can provide formal guarantees for their approximate circuit's quality constraints, and consumers are enabled to verify these constraints without having to trust the producer or transmission channels. Importantly, consumers establish trust at a fraction of the computational effort needed for full verification. We have outlined two scenarios for proof-carrying AxCs with different producer-consumer relations, and experimentally demonstrated and validated the feasibility of proof-carrying AxCs.

7

SEARCH SPACE CHARACTERIZATION FOR APPROXIMATE LOGIC SYNTHESIS

7.1 OVERVIEW

In this dissertation, we have defined four main steps for a search-based ALS process (see Section 2.1): search, approximate, verify, and estimate (or evaluate). We have described the search as the central control unit, which explores the search space of approximate circuits (AxCs) and invokes approximation techniques, quality assurance methods, and synthesis on-demand to generate, verify, or evaluate an AxC, resulting in a search–generate–verify–evaluate cycle. Especially when formal verification is employed, the verification step can significantly contribute to the overall runtime of the approximate logic synthesis (ALS) (see Sections 3.5.2 and 4.9.3), which practically limits ALS' capabilities to explore the space of possible solutions rapidly and/or thoroughly.

This chapter presents a formal verification-based pre-processing methodology that renders verification during ALS obsolete. Our methodology characterizes the search space *a priori* into a region that is guaranteed to contain valid AxCs regarding quality and a region for which validity is yet unknown. Our technique employs formal verification on a novel approximation miter that relates the global quality constraint of the circuit to the candidates' local error bounds, which are typically known, e.g., through error annotations in an approximate component library. Thus, our technique is independent of the actually chosen approximation technique. Subsequent ALS can then omit costly verifications and save runtime when searching in a region of the search space that has been characterized as valid. However, these reductions in formal verification runs have to be balanced against the additional verifications needed in the preceding search space characterization phase.

In summary, the major contributions of the methodology presented in this chapter are:

- We present an approach that characterizes the search space and determines local quality constraints for components of combinational circuits *prior* to approximate logic synthesis and independent of actual approximation methods.

- We have implemented our method and present experiments with several benchmark circuits that demonstrate the usefulness of our approach. Compared to a standard ALS flow, our approach adds search space characterization as a pre-processing step and achieves overall speed-ups of up to 3.7x due to distinctly fewer verifications during ALS.
- The implementation of our methodology is open-source and publicly available[1].

The remainder of the chapter is structured as follows: Section 7.2 analyzes related work and introduces our novel approach. Section 7.3 discusses our methodology, including the approximation miter and a heuristic algorithm to characterize the search space. Our experimental results are discussed in Section 7.4, and Section 7.5 concludes the chapter.

We have presented our work-in-progress at a workshop [109]. However, the methodology and the implementation considerably matured over time, and thus, this chapter relates only partially to the workshop version. In fact, the matured approach is accepted for a conference publication [107], and this chapter follows and largely cites the conference publication.

My colleague, Tobias Wiersema, developed the concept of the approximation miter. My contribution to the work comprises of the development of the methodology and its implementation. In addition, my student research assistant, Lucas Reuter, included the methodology in an automated tool flow that based on the outcome of the System Design Team Project [36] that I have supervised.

7.2 RELATED WORK AND NOVEL APPROACH

ALS starts from an input circuit with user-defined global quality constraints (GQCs) imposed upon the design's primary outputs (POs). A GQC can be formulated as $GQC := \epsilon(AxC) \leqslant T_{GQC}$ with $\epsilon(AxC)$ being the AxC's error at the PO(s), e.g., using the worst-case (WC) error metric, and T_{GQC} being the error bound. Initially, the input design is analyzed for the candidates to be approximated, which can be complex processing units, arithmetic components, or cuts in a gate level netlist.

Figure 7.1 shows an example circuit where boxes indicate six primary inputs (PIs) and one primary output, respectively, and the set of nodes $O = \{C_0, C_1, C_2, C_3, C_4\}$ models the components or operations in the design. For both components in the given candidate set $C_{set} = \{C_0, C_1\}$ there are approximated versions available in a library. Each candidate can thus implement its original operation or one of the eight available approximated versions, resulting in a total number of possible AxCs $N_{AxCs} = \prod_{c \in C_{set}} N_{Comps}(c) = 9^2 = 81$, with $N_{Comps}(c)$ being the number of available implementations for candidate $c \in C_{set}$ (cf. Equation (3.1) in Section 3.4.2).

1 https://git.uni-paderborn.de/vegaxc/vegaxc

The most often used approach in ALS is to set up a search or an optimization problem and execute a search–generate–verify–evaluate cycle to explore the search space starting from the original, exact design and iteratively expanding towards AxCs with lower quality yet potentially higher improvements in terms of the target metric. In our example, the first iteration could explore and generate the AxCs $(\mathrm{id}_0, \mathrm{C1}_{\mathrm{Orig}})$ and $(\mathrm{C0}_{\mathrm{Orig}}, \mathrm{id}_0)$, i.e., the AxCs where each candidate implements the component with the next-lowest quality from the library.

While the target metric is sought to be optimized, the quality states an user-defined constraint and thus has to be satisfied to represent a valid AxC. Although the quality of the individual approximated candidates is known, the resulting quality at the design's PO(s) is unknown, as the correlation between the local error at the candidates and the global error of the design can be quite involved depending on the actual circuit and its types of operations. Consequently, the verification step is inevitable to determine the AxC's validity. As verifications are usually very costly [90, 102], ALS can endure long runtimes or has to spare a thorough search space exploration.

In fact, for the verification, the vast majority of related work focuses on the error analysis of generated AxCs, i.e., the quality verification is done *a posteriori* to generating the AxC. MACACO [95] computes various error metrics for AxCs subjected to functional and timing-induced approximations. Chandrasekharan et al. [23], Vašíček [90], and Abed et al. [1] employ formal methods to determine errors of AxCs precisely. While Vašíček [90] and Abed et al. [1] consider adders and multipliers in their experiments, Chandrasekharan et al. [23] analyze different error metrics for more complex sequential AxCs; some experiments, however, failed due to the problem's complexity. Sengupta et al. [80] presented a method to determine an AxC's error by propagating the probability mass functions of the individual candidates to the AxC's primary output. The candidates, however, are limited to adders and multipliers.

We propose a novel approach to ALS that characterizes the search space *a priori* to the search–generate–verify–evaluate cycle. The approach utilizes formal verification and aims at identifying a part of the search space for which we can guarantee that it contains only valid AxCs; the validity of the AxCs in the residual search space remains unknown. Through this *a priori*

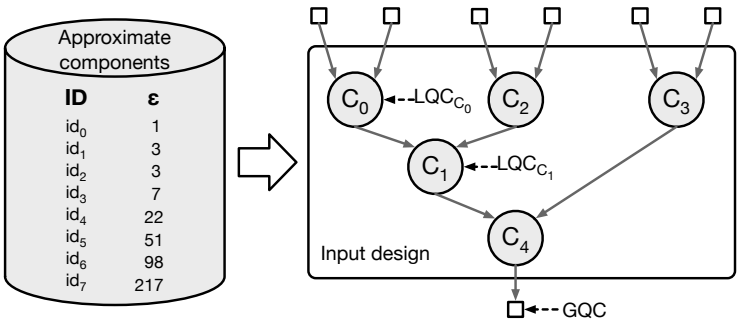


Figure 7.1: Exemplary design with approximate components. Taken from [107].

knowledge, ALS can omit verifications, which is desirable for ALS runtime reduction [102].

Instead of performing verifications on concrete candidate versions, we reduce each candidate to the error at their local outputs since this is their only relevant property regarding our characterization. Similar to the global quality constraint for an AxC, we introduce local quality constraints (LQCs) for each candidate. As already mentioned, whether a given candidate adheres to a specific LQC is generally known, for example, through worst-case error annotations in the libraries or since the employed approximation technique provides a quality parameter. Using formal verification and a set of LQCs for the candidates, our method will ascertain the worst-case error propagation of accordingly constrained local outputs to the primary outputs of the circuit. If we can formally prove that no GQC violation exists for the set of LQCs imposed upon the candidates' outputs, then we call the conjunction of these LQCs a *valid LQC combination*. In other words, independent of approximation techniques, our method guarantees valid AxCs if each candidate adheres to its LQC defined in a valid LQC combination. Note that usually multiple valid LQC combinations exist that distribute the global error slack differently among the candidates in the design. If the valid LQC combinations are known for a given circuit and global quality constraint, ALS can simply check whether the currently selected implementations for the candidates satisfy the LQCs, instead of having to formally verify the overall design.

Assume, for the example in Figure 7.1, that it is known that the $GQC := \epsilon(AxC) \leqslant T_{GQC}$ cannot be violated if the AxC satisfies the valid LQC combination $LQC_{C_0} := \epsilon(C_0) \leqslant 4 \wedge LQC_{C_1} := \epsilon(C_1) \leqslant 4$, where $\epsilon(c)$ is the quality of candidate $c \in C_{set}$. Then, the AxC represented by $(C_0 = id_0, C_1 = id_2)$, which corresponds to the candidate qualities $\epsilon(C_0) \leqslant 1$ and $\epsilon(C_1) \leqslant 3$, is known to be valid.

Valid LQC combinations can be determined either via our approach based on formal verification or, alternatively, by analytical error analysis. TDApprox from Ansaloni et al. [5] is an example of an analytical approach. In their work, candidates are nodes in a data flow graph (DFG) representation of the input design and are limited to addition, subtraction, multiplication, and division. For these node types, the authors define a set of computations to determine the error at the node's output, given the errors at the node inputs. Using these manually-specified propagation rules, the error at the DFG's output can be expressed with the errors of the candidates. The key difference between TDApprox and our method is that TDApprox analyzes the DFG of the design, while our method operates on the logical representation of a custom approximation miter (see Section 7.3.2) to verify a satisfiability (SAT) problem.

On the one hand, operating at a higher level of abstraction, such as the DFG, potentially enables shorter runtimes; on the other hand, a higher level of abstraction sacrifices details. In fact, Vašíček [90] stated that constructing accurate yet simple mathematical models for analytical approaches at gate level is currently impossible, as discussed in Section 2.1.1. Furthermore,

TDApprox relies on a manually-specified set of error propagation rules to perform the error analysis, which limits its generality by requiring the input design to only comprise supported operations. Our verification-based method overcomes these limitations. Our approach is independent of approximation methods and spares manually-specified error propagation rules since the propagation is inherently encoded in the design's logic, which can be represented on the gate level or the register-transfer level (RTL) as we operate in the satisfiability modulo theories (SMT) domain (see Section 2.2). Consequently, our approach supports arbitrary operations. The advantage of supporting arbitrary operations becomes evident when an input design is approximated incrementally, i.e., an iteration's starting design already contains approximated (or arbitrary) operations. In this case, TDApprox requires propagation rules for the specific approximate functions.

7.3 SEARCH SPACE CHARACTERIZATION VIA FORMAL VERIFICATION

We want to characterize the search space in the sense that we identify a part of it for which we can guarantee that it contains only valid AxCs. The most important step to achieve this is to verify whether a specific set of LQCs, to which the candidates adhere, enforces the AxC to be valid. To this end, we introduce and set up a custom approximation miter that indicates GQC violations in case a solver for SMT can prove the satisfiability of the miter's formula. If the SMT solver returns unsatisfiability, the AxC is valid. In contrast to commonly used approximation miters [23], our novel miter considers no specific implementation or approximation for the candidates in the AxC. Instead, our miter exposes the outputs of the candidates as new primary inputs, turning them into free variables for the SMT solver. In other words, we disconnect the candidates' outputs from the design and give the SMT solver full control to assign – in principle – any possible bit pattern instead. Naturally, this subsumes every possible approximation technique applied to the candidates, and thus, makes our approach independent of concrete approximation techniques.

We add LQC combinations as assertions to the formula to be given to the SMT solver. These LQC combinations limit the error between the precise output values of the original candidate and the output values of the approximated candidate, which the SMT solver assigns. In this way, candidate outputs violating the LQCs are prevented from satisfying the miter, and we have thus reduced the verification problem from identifying all valid combinations of all possible candidate approximations to finding valid combinations of constraints for the newly added input signals.

In the following, we first describe the preparation of the AxC for its usage in the novel approximation miter, followed by the presentation of that miter. Finally, we discuss an algorithm for determining LQC combinations that will result in valid AxCs.

7.3.1 *Augmenting the Candidates*

Before forming the SMT formula of the approximation miter, we augment the candidates of the design by extending them with additional logic and primary inputs. Figure 7.2 shows the design from Figure 7.1 whose candidates ($C_{set} = \{C_0, C_1\}$) have been augmented to $C_{Augset} = \{C_{0_{Aug}}, C_{1_{Aug}}\}$. For $c \in C_{set}$, the signals $\epsilon(c)$ and LQC_c represent the errors of candidates and the LQCs, respectively. The primary inputs $S_c, c \in C_{set}$, are the free variables for the SMT solver that override the candidate outputs. The figure conceptualizes how the LQCs constrain the primary inputs by merging the signals of the LQC and the corresponding primary input. It has to be noted that these signals are only used in the verification environment, i.e., in the SMT formula, but not in the physical implementation of the AxC. Hence, they are indicated by dashed lines in Figure 7.2.

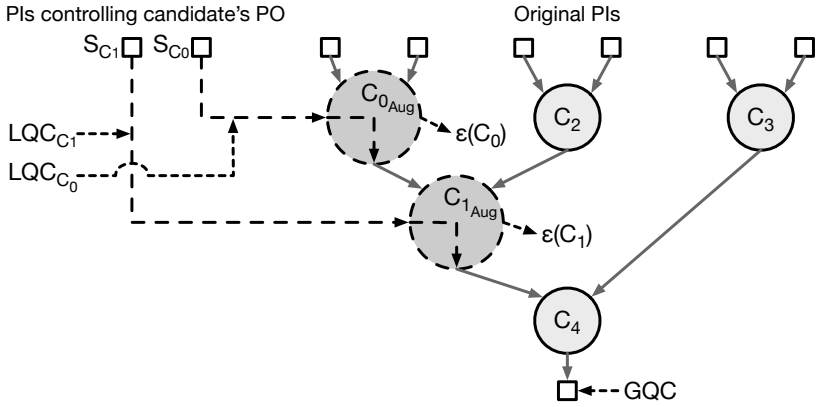


Figure 7.2: Extended example design showing the augmented candidates with their additional primary inputs and their formal signals $\epsilon(c)$ and $LQC_c, c \in C_{set}$. Taken from [107].

Figure 7.3 details the internals of the augmented candidate $C_{0_{Aug}}$ from Figure 7.2. The augmented PI of the candidate S_{C_0} is directly connected to the candidate's output. Thus, the SMT solver fully controls the candidate's functionality through setting S_{C_0}, turning the actual candidate into a black box. The original candidate C_0 is still embedded, however, and forms together with the local error computation (the WC error in the figure) the augmented candidate $C_{0_{Aug}}$. The local error computation compares the precise result of the candidate with the override signal provided by the SMT solver.

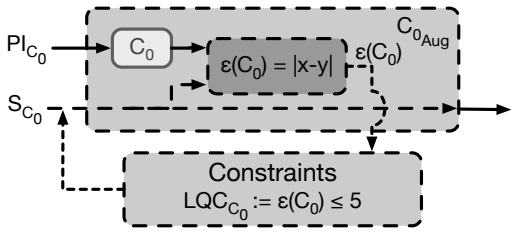


Figure 7.3: Detailed augmented candidate. Taken from [107].

The resulting error $\epsilon(c)$ is used to formulate the assertions on the LQCs and Figure 7.3 visualizes exemplarily the concept of constraining a candidate's output via an LQC. When $LQC_{C_0} := \epsilon(C_0) \leqslant 5$ is asserted in the SMT formula to constrain the candidate's WC error, it effectively forces the difference between the output of C_0 and the signal S_{C_0} to be at most 5, since the solver has to satisfy the asserted LQC in order to satisfy the SMT formula. The solver consequently excludes assignments violating the LQC since the SMT problem cannot be satisfied otherwise.

By replacing the candidates with empty black boxes and constraining the error ranges at their outputs via the LQCs in terms of the candidates' precise results, we effectively enable error propagation through the circuit in a way that captures all possible cancellation and amplification effects between the candidates for any approximations the candidates might introduce. Importantly, candidates are not limited to a certain functionality, e.g., addition or multiplication, but can implement any functionality, i.e., arithmetic or logical functions, and the augmentation only affects the candidate's internals while the circuit's general structure is maintained. In the approximation miter, the augmented design represents the AxC that is verified against the original design.

7.3.2 *Approximation miter*

Figure 7.4 shows the conceptual structure of our custom approximation miter. The miter comprises of the augmented circuit as shown in Figure 7.2, the original circuit, and a module to compute the WC error $\epsilon(AxC)$ between the outputs of the two circuits. As already described, the candidates in the augmented circuit are extended by new PIs ($S_c, c \in C_{set}$), which are directly connected to the outputs of the candidates and are free variables for the SMT solver.

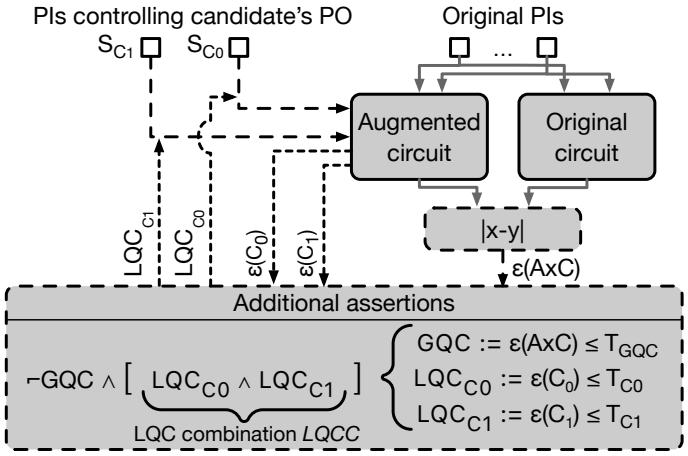


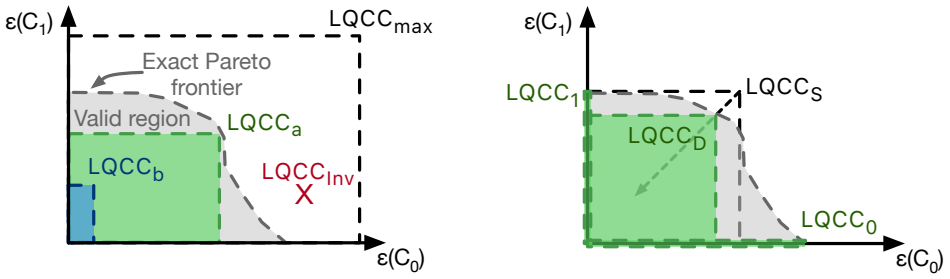
Figure 7.4: Approximation miter. Taken from [107].

Assertions add LQC combinations to the miter's formula to constrain the local errors of the candidates. The miter encodes the verification prob-

lem in the form of the logical expression $\neg GQC \wedge LQCC$, where the LQC combination LQCC is defined as the conjunction of LQCs for all candidates $LQCC := \bigwedge_{c \in C_{set}} LQC_c$.[2] Upon verification, the SMT solver seeks to find a satisfying assignment to this expression by properly setting the free variables, i.e., the original PIs and the additional PIs S_c. In order for the expression to be satisfiable, the LQC combination has to be satisfied while the GQC has to be violated. Consequently, the solver will only consider output values for the candidates that satisfy the respective LQCs in the LQC combination. If the solver proves the expression unsatisfiable, any AxC adhering to the asserted LQC combination is guaranteed to be valid, regardless of the applied approximation technique for the candidates.

7.3.3 *Search Space Characterization Algorithm*

In order to generate satisfying assignments for the approximation miter, the SMT solver deals with a search space spanned by possible assignments to the original and newly added PIs. While assigning values to the original PIs covers all possible circuit inputs, we focus the discussion on the newly added PIs, which are controlled by LQCs and corresponding assertions in the miter formula. For our considerations, we can raise the abstraction level when discussing search spaces to the local errors at which the LQCs are defined. Then, for the two candidates, C_0 and C_1, in our running example, the search space is spanned by the candidate's errors, $\epsilon(C_0)$ and $\epsilon(C_1)$, as depicted in Figure 7.5. Each point in such a space fixes the candidate's local errors and forms the root of a sub-search space in which the SMT solver is free to assume any bit pattern for the original PIs.



(a) $LQCC_a$ dominates $LQCC_b$ and $LQCC_{Inv}$ is invalid.

(b) Visualization of our algorithm.

Figure 7.5: Search space example with two candidates, C_0 and C_1. The candidates' errors, $\epsilon(C_0)$ and $\epsilon(C_1)$, define the search space's dimensions. Taken from [107].

This search space can generally be divided into a valid and an invalid region, with the valid region being the union of all possible valid LQC combinations. One observation when working with error metrics formulated as error bounds, such as the WC error, is that such metrics include, or dominate, stronger bounds inherently. For example, the WC error threshold

2 N LQCCs can be verified jointly via their disjunction $\neg GQC \wedge \bigvee_{0 \leqslant i < N} LQCC_i$.

of $\epsilon \leqslant 5$ is included in $\epsilon \leqslant 7$. Thus, when a specific LQC combination is verified to lead to a valid AxC, we know that all LQC combinations with stronger error bounds will also lead to valid AxCs. The contrary is obviously not true, i.e., an LQC combination that leads to an invalid AxC since the SMT solver can find a satisfying assignment does not give us any information about LQC combinations with stronger error bounds. Thus, a verified LQC combination characterizes a valid region of the search space as a hypercuboid of dimension $|C_{set}|$ with the origin as one edge.

Figure 7.5a shows the search space for our two candidates with the valid region indicated in gray. $LQQC_{max}$ represents the LQC combination with maximum errors for the candidates, $LQCC_{Inv}$ an invalid LQC combination, and $LQCC_a$ and $LQCC_b$ valid combinations of local error bounds. $LQCC_a$ *dominates* $LQCC_b$ since the error bounds for the LQCs in $LQCC_a$ are not smaller than the corresponding bounds in the LQC combination $LQCC_b$, and at least one LQC in $LQCC_a$ has a higher threshold than $LQCC_b$. Using this dominance concept, we can define a Pareto frontier of non-dominated LQC combinations that delineates a border between the valid and invalid regions of the search space.

Finding the exact Pareto frontier in the search space is an intractable task. Thus, we propose a simple, low-overhead algorithm that verifies a limited number of LQC combinations to keep the runtime short while still characterizing large-enough valid regions to speed up subsequent ALS. Algorithm 7.1 outlines our methodology and takes the exact circuit O, the candidate set C_{set}, and the GQC as inputs. During execution, the algorithm verifies different LQC combinations by adding the corresponding assertions to the miter and records the validity of LQC combinations.

Firstly, the algorithm augments the candidates in the design and constructs the approximation miter AM in Lines 3 and 4. Secondly, the algorithm iterates over the candidates $c \in C_{set}$ to determine a maximally allowed error for each candidate in Line 5 to Line 13. In order to determine the maximal error of each candidate, Line 6 generates an LQC for each candidate $c_z \in C_{set} \setminus c$ that enforces error-free candidates. Then, starting from a maximum error bound for a candidate, Lines 8 to 13 verify the error bounds in descending order until the SMT solver reports *unsatisfiable*. As the unsatisfiable combination dominates all remaining LQC combinations of that candidate, the valid LQC combination is added to the set of dominating LQC combinations $LQCC_{dom}$ and the algorithm continues with the next candidate. Figure 7.5b depicts the valid LQC combinations $LQCC_0$ and $LQCC_1$ of the respective candidates. Since the number of possible error bounds grows exponentially with the candidate's output bit width, Line 9 only encodes the powers of two into the LQC combinations to reduce the number of verifications and establish a fine-grained resolution for small error bounds. For example, to determine the maximally allowed error for a candidate with a 4-bit unsigned output, our algorithms employs the WC error bounds 1, 2, 4, and 8, while enforcing other candidates to be error-free.

Algorithm 7.1: Pseudo-code of our search space characterization.

Input: Exact circuit O, candidate set C_{set}, global quality constraint GQC
Output: Dominant LQC combinations $LQCCs_{dom}$

```
1  Function characterize( O, C_set, GQC):
2      LQCCs_dom ← ∅                          // set storing dominant LQC combinations
3      AxC ← augmentCandidates( O, C_set )
4      AM ← constructApproximationMiter( O, AxC, GQC )
       /* Find maximum error of the individual candidates                    */
5      foreach c ∈ C_set do
6          LQCC_z ← ⋀_{c_z ∈ C_set \ c} ε(c_z) == 0          // enforce zero error
7          bw ← bitwidth(c) − 1
8          while bw ⩾ 0 do
9              LQCC ← LQCC_z ∧ ε(c) ⩽ 2^bw          // consider power of two
10             if verify( AM, LQCC ) == unsatisfiable then
11                 LQCCs_dom ← LQCCs_dom ∪ LQCC
12                 break
13             bw ← bw − 1
       /* Find maximum errors on diagonal of hypercube                      */
14     bw ← min(log2(LQCC_dom))    // find min. error bound among candidates
15     while bw ⩾ 0 do
16         LQCC ← ⋀_{c ∈ C_set} ε(c) ⩽ 2^bw
17         if verify( AM, LQCC ) == unsatisfiable then
18             LQCCs_dom ← LQCCs_dom ∪ LQCC
19             break
20         bw ← bw − 1
       /* Return dominating LQC combinations                                */
21     return LQCCs_dom
```

Lastly, our algorithm considers the hypercube with edge length set to the minimum of all maximally allowed error bounds for the single candidates in Line 14, and verifies LQC combinations along the hyper diagonal in descending order of error bounds from Line 15 to 20. For example, for the two candidates shown in Figure 7.5b, the maximally allowed error bound of C_1 is smaller than that of C_0, and thus, the search starts at $LQCC_S$ and finds the dominating valid LQC combination in $LQCC_D$. Line 16 limits again the error bounds to powers of two to reduce complexity. Upon termination, the algorithm returns the set of dominating LQC combinations LQC_{dom}.

The choice of searching the diagonal of the hypercube is a heuristic, and naturally more involved approaches exist that more accurately characterize the search space. However, the presented approach is efficient as it performs at most $\min_{c \in C_{set}}(bw(c)) + \sum_{c \in C_{set}} bw(c)$ verifications with $bw(c)$ being the bit width of candidate $c \in C_{set}$, and returns $(|C_{set}| + 1)$ valid LQC combinations. The search–generate–verify–evaluate cycle of subsequent ALS can now omit the verification step for the combinations of local error bounds that are dominated by any of these valid LQC combinations.

7.4 EXPERIMENTAL RESULTS

We have implemented our methodology in an automated open-source tool flow which expects a RTL or gate level input design in Verilog. Initially, our flow augments the candidates and constructs the approximation miter using custom scripts and Yosys [110] passes. A Python script implements our algorithm from Section 7.3.3 and translates the miter into SMT-LIBv2 [11] standard format. The flow then utilizes Yosys' smt2 back-end to interface the SMT solver Boolector [66], which performs in incremental mode and utilizes CaDiCaL [16] as underlying SAT solver.

Subsequent to our flow, we employ the CIRCA framework (see Chapter 3) to perform a search-based ALS using hill climbing search for search space exploration, precision scaling and AIG rewriting [22] as approximation techniques, and ABC's [14] *if* command for look-up table mapping to evaluate the hardware area. Our ALS setup omits verifications for AxCs that are known to lie within the valid portion of the search space; otherwise, ALS invokes Boolector to check the AxC's validity.

Our benchmark set comprises of four circuits of which one was altered in the number of candidates to evaluate scalability, resulting in six benchmarks in total. We have selected arithmetic components as candidates and used the worst-case error as global quality metric, which we varied from 0.5% to 10.0% of the maximal possible output value. A server with 32 Gigabyte main memory and an Intel® Xeon E5-2609@2.5GHz has been employed to run the experiments.

Figure 7.6 compares experimental results for the standard ALS (denoted `ALS`) with the ALS that uses our methodology as pre-processing step (denoted `Our`), where the orange bar indicates the portion of the time spent on search space characterization. The plot reveals that performing our methodology prior to ALS is beneficial for all but one combination of benchmark and WC error. We observe speed-ups of up to 3.7× (cf. `mac`, WC error 10.0%) and, for example, for `basic_sad_4` with a WC error of 2.5%, the runtime decreased from 480s to 240s. The time required for search space characterization ranges between 1% and 38% of the total runtime and averages over all experiments to ≈11% of the total runtime. Most importantly, it can be seen that this additional time is compensated during ALS by omitting verifications. Merely for `basic_sad_6` with a WC error of 2.5%, the total runtime increases by 113s since the verification in the characterization phase shows exceptionally long runtimes; in fact, a single SMT query caused the increased runtime in this case by accounting for 360s of the 410s of the characterization phase's runtime.

Table 7.1 details our experimental results and shows for each benchmark the number of original PIs and POs, the number of candidates, the number of added PI bits for verification, and the total number of possible LQC combinations in the search space. Furthermore, the table lists for each benchmark and WC error combination the number of verifications performed during ALS: first the number of verifications skipped due to our methodology, then

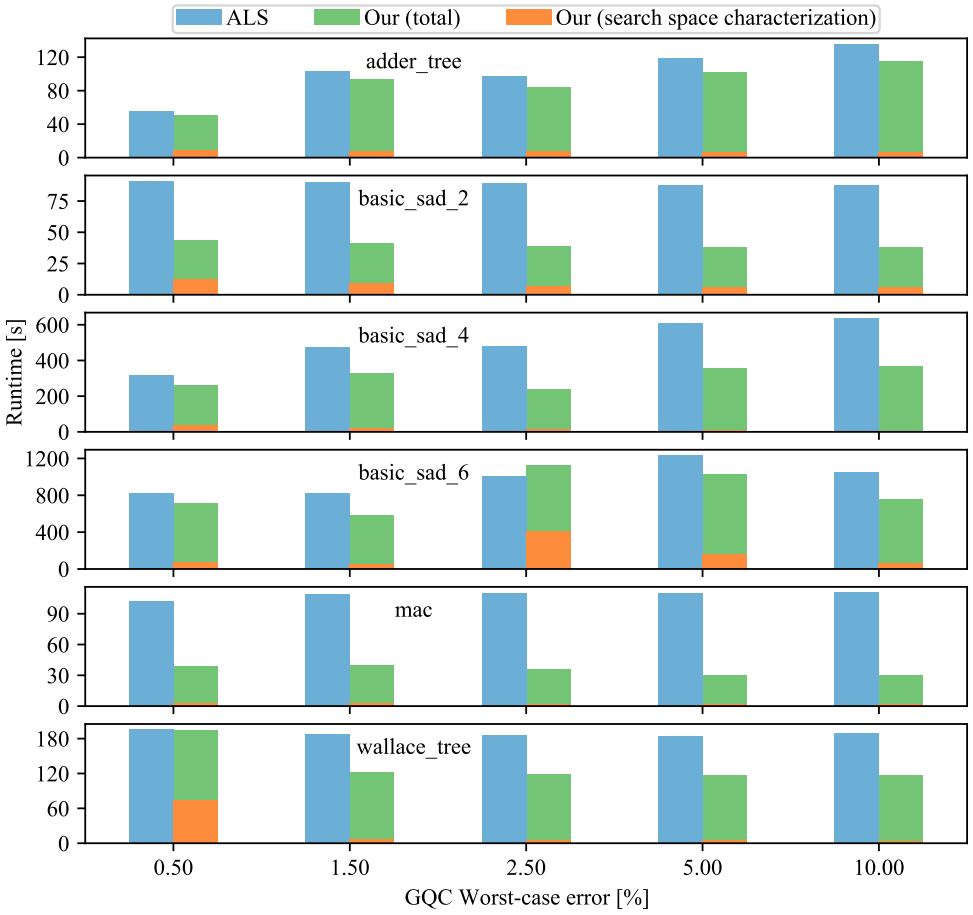


Figure 7.6: Runtimes of ALS and our methodology. Taken from [107].

the number of verifications performed resulting in valid AxCs, and, finally, the total number of verifications. We can interpret the ratio between skipped verifications and valid AxCs as an indicator for our method's coverage of valid AxCs on the search path taken by the search method. For our benchmarks, this ratio ranges between 50% and 100%. Thus, we conclude that, even though our search space characterization algorithm is kept simple, the coverage is sufficient to achieve significant reductions in runtime.

The achieved speed-up depends on the benchmark and the corresponding difficulty it poses for verification, the WC error bound, and also on the number of candidates. For analyzing scalability with the number of candidates for a given benchmark, we varied the number of candidates for `basic_sad`. As expected, more candidates improve the area savings at the cost of increased complexity, indicated by increasing numbers in Figure 7.6 and Table 7.1. However, for all numbers of candidates, we have been able to achieve speed-ups for all but one WC error bound. In fact, our approach renders 55% to 100% of the verifications obsolete for the valid AxCs, reducing the runtime by 55%, 36%, and 15% on average for two, four, and six candidates, respectively.

Additionally, Table 7.1 displays the number of synthesized AxCs. As it can be seen, the number of synthesized AxCs is generally much larger

Table 7.1: Experimental results. Taken from [107].

GQC: WC bound [%]	**0.50**	**1.50**	**2.50**	**5.00**	**10.00**
adder_tree (*PIs:* 8 × 8 *bit; PO:* 11 *bit;* 3 *candidates*)					
#add. PI bits: 27*; Total #LQC combs.:* 1.34×10^8					
#Verifications†	4 / 6 (10)	5 / 11 (21)	6 / 12 (19)	7 / 15 (24)	8 / 18 (28)
#Syn. AxCs	26	41	44	53	61
Rel. area [%]	88	79	74	68	62
#LQC combs.	45	753	4961	36,033	2.75×10^5
basic_sad_2 *(PIs:* 18 × 8 *bit; POs:* 13 *bit;* 2 *candidates)*					
#add. PI bits: 22*; Total #LQC combs.:* 4.19×10^6					
#Verifications†	4 / 4 (4)	4 / 4 (4)	4 / 4 (4)	4 / 4 (4)	4 / 4 (4)
#Syn. AxCs	12	12	12	12	12
Rel. area [%]	98	98	98	98	98
#LQC combs.	321	1153	4353	16,897	66,305
basic_sad_4 *(PIs:* 18 × 8 *bit; PO:* 13 *bit;* 4 *candidates)*					
#add. PI bits: 43*; Total #LQC combs.:* 8.80×10^{12}					
#Verifications†	6 / 11 (12)	11 / 18 (19)	16 / 18 (19)	17 / 24 (26)	18 / 26 (27)
#Syn. AxCs	60	88	88	107	113
Rel. area [%]	91	87	86	79	78
#LQC combs.	6657	83,713	1.19×10^6	1.79×10^7	2.77×10^8
basic_sad_6 *(PIs:* 18 × 8 *bit; PO:* 13 *bit;* 6 *candidates)*					
#add. PI bits: 61*; Total #LQC combs.:* 2.31×10^{18}					
#Verifications†	10 / 17 (28)	17 / 21 (26)	18 / 29 (33)	23 / 34 (46)	24 / 35 (35)
#Syn. AxCs	166	190	237	258	259
Rel. area [%]	87	83	77	71	65
#LQC combs.	15,793	2.41×10^7	1.29×10^9	7.54×10^{10}	4.61×10^{12}
mac *(PIs:* 2 × 8 *bit,* 1 × 16 *bit; PO:* 17 *bit;* 2 *candidates)*					
#add. PI bits: 33*; Total #LQC combs.:* 8.59×10^9					
#Verifications†	13 / 14 (15)	14 / 15 (16)	15 / 16 (16)	16 / 16 (16)	16 / 16 (16)
#Syn. AxCs	32	34	36	36	36
Rel. area [%]	87	86	86	86	86
#LQC combs.	66,561	2.64×10^5	1.05×10^6	4.20×10^6	1.68×10^7
wallace_tree *(PIs:* 2 × 8 *bit; PO:* 16 *bit;* 4 *candidates)*					
#add. PI bits: 32*; Total #LQC combs.:* 4.29×10^9					
#Verifications†	18 / 25 (25)	18 / 25 (25)	18 / 25 (25)	18 / 25 (25)	18 / 25 (25)
#Syn. AxCs	116	116	116	116	116
Rel. area [%]	91	91	91	91	91
#LQC combs.	173	189	221	285	397

† #verifications skipped / #verifications resulting in valid AxCs (#total verifications).

than the number of verifications since the hill climbing search expands the search space by synthesizing all successor AxCs of a given AxC and then only performing verifications until the valid successor with the largest area reduction is found. Despite the high number of synthesis steps for evaluating AxCs, it is the verification time that dominates the overall runtime; our reduced runtimes reflect this, since we achieve the reduced runtimes by omitting verifications only.

The rows *#LQC combs.* show the total number of valid LQC combinations (dominating and non-dominating) that have been identified during search space characterization, i.e., the hypervolume of the portion of the search space characterized as valid. Taking this number as an estimate for the search space's real valid region and comparing it to the number of synthesized AxCs, we can observe that the hill climbing search only explores a fraction of the valid search space and even gets stuck rather early in local minima for the benchmarks `mac` and `wallace_tree`.

For the benchmarks where hill climbing found a local minimum, we experimented with a simple yet effective ad hoc strategy to overcome the local minima to improve the achieved savings. We followed jump search's (see Chapter 4) assumption, that a higher degree of approximations leads to improved savings, and generated AxCs from the valid LQC combinations directly. Table 7.2 shows the results of the experiment and demonstrates that the ad hoc approach reduces the area significantly while even achieving lower overall runtimes compared to hill climbing due to fewer synthesis and verification steps. The experiment thus shows that our approach allows to quickly assess potential in savings by sub-sampling the valid search space.

Table 7.2: Experimental results for generating the AxCs from the valid LQC combinations.

WC bound [%]	**0.50**	**1.50**	**2.50**	**5.00**	**10.00**	**0.50**	**1.50**	**2.50**	**5.00**	**10.00**
			mac					**wallace_tree**		
Runtime [s]	40	46	39	24	20	122	56	54	53	52
#Syn. AxCs	13	13	13	13	13	65	65	65	65	65
Rel. area [%]	84	83	81	61	49	79	79	79	78	72

7.5 CONCLUSION

In this chapter, we have presented a formal verification-based methodology to determine combinations of local quality constraints for candidates that guarantee valid approximate circuits. Our approach is independent of employed approximation methods and subsequent approximate logic synthesis flows can exploit the provided information to reduce runtime by omitting costly verifications. We have detailed our method and shown experimental results that demonstrate the benefits of an a priori search space characterization. Compared to a standard search-based ALS flow performing a hill

climbing search, we achieve speed-ups of up to 3.7× using our methodology. For the benchmarks that got stuck in local minima, we furthermore exploited concepts from jump search to improve the achieved area savings while simultaneously reducing the runtime; thus, demonstrating the capabilities of our methodology for exploring the search space efficiently.

8

CONCLUSION

Research from academia and industry has shown that approximate computing is a practical design paradigm to react to the slow-down and the eventual end of performance improvements through technology scaling. Approximate computing exploits the inherent resilience of many applications against inaccuracies and errors by trading off an application's quality against a target metric to gain performance improvements. Furthermore, the employed approximation methods are complementary and can be applied at all levels of the computing stack – from applications and programming languages down to semiconductor technology – to all system components – processing units, storage as well as communication components.

This dissertation has focused on approximate computing on the hardware level, where approximate computing is referred to as approximate logic synthesis (ALS) to describe the process of generating approximate hardware components or approximate circuits (AxCs). We have comprehensively considered automated search-based ALS at register-transfer level and logic level and have modeled the ALS process with four main steps: *search*, *approximate*, *verify*, and *estimate*. Figure 8.1 visually concludes this dissertation and highlights the contributed methodologies, techniques, and approaches to the ALS process.

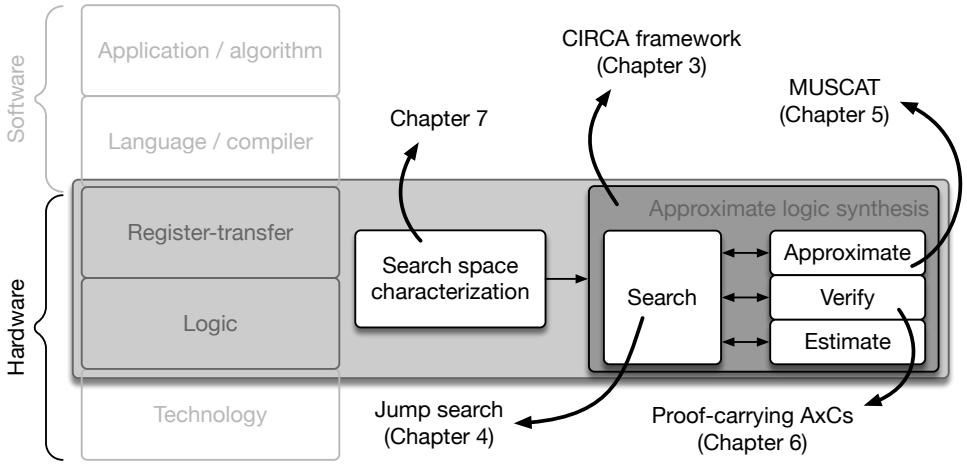


Figure 8.1: Overview of approximate logic synthesis in the computing stack and the dissertation's contributions.

In more detail, this dissertation presented the following methodologies and approaches:

- We have presented CIRCA, our general, modular, compatible, extensible, and open-source ALS framework. CIRCA's design is flexible, fully configurable, and provides an environment for implementing and evaluating different methods in any of the steps of the ALS process. We have demonstrated that CIRCA's flexibility fosters comparative studies and contributes a foundation for ALS research in general; in fact, CIRCA has provided the technical foundation for the research in this dissertation.
- We have presented jump search as a methodology to significantly reduce the runtime of approximate logic synthesis. Jump search first acquires candidate-specific information in the form of *impact factors* in a pre-processing phase. Secondly, jump search plans a path through the search space to select a set of AxCs-of-interest using a heuristic function that incorporates the impact factors and completely omits evaluations via synthesis and verification. Finally, jump search performs a binary search on the AxCs-of-interest to minimize the number of invoked evaluations, which results in a significantly reduced runtime; yet our experiments have shown that the achieved area savings remain comparable to state-of-the-art approaches. With jump search, we have demonstrated that incorporating meaningful information into the search significantly impacts the ALS process and can considerably reduce the workload.
- We have proposed the approximation technique MUSCAT that augments a gate level netlist by *cutpoints* and utilizes the concept of minimal unsatisfiable subsets to simplify the netlist's logic to create AxCs that are *valid-by-construction* regarding their quality constraints. MUSCAT has shown that modern formal verification engines can be leveraged for finding optimal solutions during ALS which outperform state-of-the-art methods. Furthermore, we have demonstrated in a case study that MUSCAT also qualifies as an approximation technique at register-transfer level to approximate complex designs at significantly lower runtimes due to the raised level of abstraction.
- With the concept of proof-carrying approximate circuits, we have combined the fields of approximate computing and proof-carrying hardware to enable consumers of approximate intellectual property (IP) cores to verify the core's quality at a fraction of the runtime of a full formal verification. In this way, consumers overcome trust issues against the producer of the IP core and have a guarantee about the quality of the purchased IP core. This dissertation has detailed the concept and the experimental results have demonstrated that proof-carrying AxCs significantly reduces the consumer's verification workload.
- Finally, we have presented a novel methodology to characterize the search space *prior to* ALS. Our formal verification-based methodology

identifies regions in the search space that are guaranteed to contain valid AxCs only, i.e., AxCs that satisfy the quality constraints. The approach is independent of the subsequently employed approximation techniques and allows ALS to omit costly verifications to reduce overall runtime, as our experimental results have shown.

9

OUTLOOK

The research in this dissertation has investigated different aspects of approximate logic synthesis (ALS) and has revealed potential future directions. The future directions are manifold and concern the individual work and the complementation of work:

- Common future work among the research projects includes investigating different error metrics, as this dissertation mainly considered the worst-case error metric. Especially when formal verification engines further advance and offer improved performance, more complex error metrics can be considered for real-world applications – maybe even average (or statistical) error metrics.
- A few research groups [25, 113, 119] investigated complementary ALS over multiple levels of abstraction. However, the conducted work only considers a limited set of approximation techniques and a unidirectional execution flow, i.e., backtracking to revoke approximation decisions on higher levels is not supported. Furthermore, identifying suitable candidates on the different abstraction levels and selecting matching approximation techniques remain open questions.

  A student project group that I supervised [32] laid a foundation in the direction of cross-layer ALS by designing and implementing a CIRCA-based ALS setup. The designed tool flow is fully configurable, spans all hardware levels, supports backtracking, and allows for analysis methods in-between ALS at different levels, e.g., to identify candidates or acquire circuit-specific information to guide ALS at the next abstraction level. The first experimental results followed the previously published work in this field and showed that complementary ALS is indeed beneficial. However, the ALS flow lacks a comprehensive methodology for identifying candidates, selecting approximation techniques best suited for a candidate, or controlling the backtracking to revoke previous ALS decisions.

  Jump search's impact factors, for example, could guide candidate selection, and, in fact, initial experiments suggest that pruning the set of candidates based on the impact factors can reduce ALS' complexity without sacrificing significant improvements [21]. The selection of a suitable approximation technique for a candidate requires a thorough evaluation of existing approximation techniques to identify techniques that work best for a given candidate at a specific level of abstraction.

- Future work in jump search includes investigating new impact factors on different metrics and low-cost methods for determining them, e.g., impact factors for power or energy consumption.

  Additionally, planning multiple paths through the search space could improve the outcome's performance. However, parallel processing of the different paths is desirable to maintain short runtimes.

  Our early research considered a post-processing phase in jump search that utilizes the remaining time budget to fine-tune the approximate circuit (AxC). However, the performed experiments have demonstrated that the additional effort could be spared since, in most cases, the runtime increased significantly while the achieved improvements stagnated.
- For MUSCAT, our conducted case study on raising the level of abstraction to the register-transfer level (RTL) has shown promising results and exposed the performance differences among formal verification engines. Thus, future work includes assessing available formal verification engines and evaluating MUSCAT's full potential at RTL by conducting comprehensive experiments with complex designs.
- Our methodology for formal verification-based search space characterization shows a new direction of research to simplify subsequent ALS, but, currently, the formal verification forms the bottleneck of this approach. In fact, the verification problem can state a complex task that increases with the design's complexity and the number of candidates. Thus, to reduce the verification complexity, an iterative approach could be employed that characterizes the search space for a subset of the candidates that are then approximated subsequently. The partially approximated circuit then forms the next iteration's starting point. Initial experiments in this direction have shown promising results to keep runtimes reasonable and thus motivate the development to enable the characterization of more complex designs.

  Additional future work for the search space characterization comprises the evaluation of search algorithms to effectively explore the search space's valid region and the support of sequential circuits – which is non-trivial. Finally, we envision the approach for runtime-configurable circuits to define the search space regions that can be operated safely at runtime. For example, MUSCAT could augment a circuit with cutpoints and determine configurations for different qualities. At runtime, a controller could then decide on the required quality and enforce the local quality constraints by applying a cutpoint configuration accordingly.

BIBLIOGRAPHY

[1] Sa'ed Abed, Ali A. M. R. Behiry, and Imtiaz Ahmad. "Error Metrics Determination in Functionally Approximated Circuits Using SAT Solvers." In: *PLOS ONE* 15.1 (Jan. 2020), pp. 1–19. DOI: 10.1371/journal.pone.0227745.

[2] Anant Agarwal, Martin C. Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures*. Tech. rep. Sept. 2009. URL: http://hdl.handle.net/1721.1/46709.

[3] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Proof-Carrying Hardware Versus the Stealthy Malicious LUT Hardware Trojan." In: *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*. Vol. 11444. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 127–136. DOI: 10.1007/978-3-030-17227-5_10.

[4] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram. "RAP-CLA: A Reconfigurable Approximate Carry Look-Ahead Adder." In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.8 (Aug. 2018), pp. 1089–1093. DOI: 10.1109/TCSII.2016.2633307.

[5] Giovanni Ansaloni, Ilaria Scarabottolo, and Laura Pozzi. "Judiciously Spreading Approximation Among Arithmetic Components with Top-Down Inexact Hardware Design." In: *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*. Vol. 12083. Lecture Notes in Computer Science. Springer, 2020, pp. 14–29. DOI: 10.1007/978-3-030-44534-8_2.

[6] Muhammad Awais. "MCTS-based Approximator Accelerator Synthesis." PhD thesis. Paderborn, Germany: Paderborn University, Apr. 2021. DOI: 10.17619/UNIPB/1-1166.

[7] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "A Hybrid Synthesis Methodology for Approximate Circuits." In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2020, pp. 421–426. DOI: 10.1145/3386263.3406952.

[8] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "LDAX: A Learning-based Fast Design Space Exploration Framework for Approximate Circuit Synthesis." In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2021, pp. 27–32. DOI: 10.1145/3453688.3461506.

[9] Muhammad Awais and Marco Platzner. "MCTS-based Synthesis Towards Efficient Approximate Accelerators." In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2021, pp. 384–389.

[10] Mario Barbareschi, Federico Iannucci, and Antonino Mazzeo. "Automatic Design Space Exploration of Approximate Algorithms for Big Data Applications." In: *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE. 2016, pp. 40–45. DOI: 10.1109/WAINA.2016.172.

[11] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. [Online]. 2016. URL: https://smtlib.cs.uiowa.edu/.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications 1. IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825.

[13] Jaroslav Bendík and Ivana Cerná. "MUST: Minimal Unsatisfiable Subsets Enumeration Tool." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 135–152. DOI: 10.1007/978-3-030-45190-5_8.

[14] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. [Online]. URL: https://github.com/berkeley-abc/abc.

[15] Armin Biere, Tom van Dijk, and Keijo Heljanko. "Hardware Model Checking Competition 2017." In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2017, p. 9. DOI: 10.23919/FMCAD.2017.8102233.

[16] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020." In: *Proceedings of the SAT Competition 2020 – Solver and Benchmark Descriptions*. Vol. B-2020-1. Dept. of Comput. Sci. Rep. Ser. B. University of Helsinki, 2020, pp. 51–53.

[17] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. "Uncertain<T>: Abstractions for Uncertain Hardware and Software." In: *IEEE Micro* 35.3 (2015), pp. 132–143. DOI: 10.1109/MM.2015.52.

[18] Jorge Castro-Godínez, Sven Esser, Muhammad Shafique, Santiago Pagani, and Jörg Henkel. "Compiler-driven Error Analysis for Designing Approximate Accelerators." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1027–1032. DOI: 10.23919/DATE.2018.8342163.

[19] Milan Češka, Jiří Matyáš, Vojtěch Mrázek, Lukáš Sekanina, Zdeněk Vašíček, and Tomáš Vojnar. "Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-Bit Multipliers Accomplished." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 416–423. DOI: 10.1109/ICCAD.2017.8203807.

[20] Milan Češka, Jiří Matyáš, Vojtěch Mrázek, Lukáš Sekanina, Zdeněk Vašíček, and Tomáš Vojnar. "ADAC: Automated Design of Approximate Circuits." In: *Computer Aided Verification*. Springer, July 2018, pp. 612–620. DOI: 10.1007/978-3-319-96145-3_35.

[21] Khushboo Chandrakar. "Comparison of Feature Selection Techniques to Improve Approximate Circuit Synthesis." Master's Thesis. Paderborn, Germany: Paderborn University, Sept. 2020.

[22] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. "Approximation-aware Rewriting of AIGs for Error Tolerant Applications." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM, 2016. DOI: 10.1145/2966986.2967003.

[23] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. "Precise Error Determination of Approximated Components in Sequential Circuits with Model Checking." In: *Proceedings of the Design Automation Conference (DAC)*. ACM Press, 2016. DOI: 10.1145/2897937.2898069.

[24] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. "Analysis and Characterization of Inherent Application Resilience for Approximate Computing." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2013. DOI: 10.1145/2463209.2488873.

[25] Vinay Kumar Chippa, Debabrata Mohapatra, Kaushik Roy, Srimat T. Chakradhar, and Anand Raghunathan. "Scalable Effort Hardware Design." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.9 (2014), pp. 2004–2016. DOI: 10.1109/TVLSI.2013.2276759.

[26] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 7795. LNCS. Springer, 2013.

[27] Altera Corporation. *Altera Advanced Synthesis Cookbook*. [Online]. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf.

[28] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II." In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[29] Stephanie Drzevitzky. "Proof-Carrying Hardware : A Novel Approach to Reconfigurable Hardware Security." PhD thesis. Paderborn, Germany: Paderborn University, 2012. URL: http://nbn-resolving.de/urn:nbn:de:hbz:466:2-10423.

[30] Stephanie Drzevitzky, Uwe Kastens, and Marco Platzner. "Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification." In: *International Journal of Reconfigurable Computing* 2010 (2010). DOI: 10.1155/2010/180242.

[31] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient Implementation of Property Directed Reachability." In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.

[32] Sebastian Eikenberg, Kai Arne Hannemann, Aniruddh Rao, Oliver Reimer, and Simon Thiele. *CIRCA: An Approximate Computing Tool Flow*. Tech. rep. Paderbon University, 2021. URL: https://en.cs.uni-paderborn.de/ceg/teaching/student-projects/project-groups/circa-an-approximate-computing-tool-flow.

[33] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Architecture Support for Disciplined Approximate Programming." In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2012, pp. 301–312. DOI: 10.1145/2150976.2151008.

[34] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-Purpose Approximate Programs." In: *IEEE Micro* 33.3 (2013), pp. 16–27. DOI: 10.1109/MM.2013.28.

[35] Saman Fröhlich, Daniel Große, and Rolf Drechsler. "Error Bounded Exact BDD Minimization in Approximate Computing." In: *International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE Computer Society, 2017, pp. 254–259. DOI: 10.1109/ISMVL.2017.11.

[36] Abdulkarim Ghazal, Jonas Hölscher, Roman Meusch, Lucas Reuter, Harun Sahin, Martin Schröder, and Michael Siegmund. *Approximate Computing: The Design of Intentionally Incorrect Digital Hardware*. Tech. rep. Paderborn, Germany: Paderbon University, 2021. URL: https://en.cs.uni-paderborn.de/ceg/teaching/student-projects/system-design-team-project/approximate-computing-the-design-of-intentionally-incorrect-digital-hardware.

[37] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. "Resilience Evaluation for Approximating SystemC Designs Using Machine Learning Techniques." In: *Proceedings of the International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2018, pp. 97–103. DOI: 10.1109/RSP.2018.8631997.

[38] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. "IMPACT: Imprecise Adders for Low-Power Approximate Computing." In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*. IEEE/ACM, 2011,

pp. 409–414. URL: http://portal.acm.org/citation.cfm?id=2016898&CFID=34981777&CFTOKEN=25607807.

[39] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. "Minimal Unsatisfiable Core Extraction for SMT." In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016, pp. 57–64. DOI: 10.1109/FMCAD.2016.7886661.

[40] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. "On-the-Fly Computing: A Novel Paradigm for Individualized IT Services." In: *Proceedings of the International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*. IEEE. IEEE, 2013. DOI: 10.1109/ISORC.2013.6913232.

[41] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. "BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2018. DOI: 10.1145/3195970.3196001. arXiv: 1805.06050v1 [cs.AR]. URL: http://arxiv.org/abs/1805.06050v1.

[42] Jiawei Huang, John C. Lach, and Gabriel Robins. "A Methodology for Energy-Quality Tradeoff Using Imprecise Hardware." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2012, pp. 504–509. DOI: 10.1145/2228360.2228450.

[43] Tobias Isenberg, Marco Platzner, Heike Wehrheim, and Tobias Wiersema. "Proof-Carrying Hardware via Inductive Invariants." In: *ACM Transactions on Design Automation of Electronic Systems* 22.4 (July 2017), 61:1–61:23. DOI: 10.1145/3054743.

[44] Honglan Jiang, Jie Han 0001, and Fabrizio Lombardi. "A Comparative Review and Evaluation of Approximate Adders." In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)* (2015), pp. 343–348. DOI: 10.1145/2742060.2743760. URL: http://dl.acm.org/citation.cfm?doid=2742060.2743760.

[45] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications." In: *Proceedings of the IEEE* 108.12 (Dec. 2020), pp. 2108–2135. DOI: 10.1109/jproc.2020.3006451.

[46] C. Krieg, C. Wolf, and A. Jantsch. "Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 2016, pp. 1–8. DOI: 10.1145/2966986.2967054.

[47] Łukasz Langa. *Configuration file parser - configparser*. [Online]. 2011. URL: https://github.com/jaraco/configparser.

[48] Chaofan Li, Wei Luo, Sachin S. Sapatnekar, and Jiang Hu. "Joint precision optimization and high level synthesis for approximate computing." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2015, 104:1–104:6. DOI: 10.1145/2744769.2744863.

[49] Cong Liu, Jie Han, and Fabrizio Lombardi. "An Analytical Framework for Evaluating the Error Characteristics of Approximate Adders." In: *IEEE Transactions on Computers* 64.5 (2015), pp. 1268–1281. DOI: 10.1109/TC.2014.2317180.

[50] Gai Liu and Zhiru Zhang. "Statistically Certified Approximate Logic Synthesis." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (2017), pp. 344–351. DOI: 10.1109/iccad.2017.8203798.

[51] Eric Love, Yier Jin, and Yiorgos Makris. "Enhancing Security via Provably Trustworthy Hardware Intellectual Property." In: *Proceedings of the International Symposium on Hardware-oriented Security and Trust (HOST)*. IEEE Computer Society, 2011, pp. 12–17. DOI: 10.1109/HST.2011.5954988.

[52] David Lundgren. *OpenCores jpegencode*. [Online]. URL: https://github.com/chiggs/oc_jpegencode.

[53] Jason Luu et al. "VTR 7.0: Next Generation Architecture and CAD System for FPGAs." In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.2 (July 2014), 6:1–6:30. DOI: 10.1145/2617593. URL: http://doi.acm.org/10.1145/2617593.

[54] Hamid Reza Mahdiani, Ali Ahmadi, Sied Mehdi Fakhraie, and Caro Lucas. "Bio-inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57-I.4 (2010), pp. 850–862. DOI: 10.1109/TCSI.2009.2027626.

[55] Mahmoud Masadeh, Osman Hasan, and Sofiène Tahar. "Comparative Study of Approximate Multipliers." In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2018, pp. 415–418. DOI: 10.1145/3194554.3194626.

[56] Chang Meng, Weikang Qian, and Alan Mishchenko. "ALSRAC: Approximate Logic Synthesis by Resubstitution with Approximate Care Set." In: *Proceedings of the Design Automation Conference (DAC)*. IEEE, 2020. DOI: 10.1109/dac18072.2020.9218627.

[57] Asit K. Mishra, Rajkishore Barik, and S. Paul. "iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing." In: *Proceedings of the Workshop on Approximate Computing Across the System Stack (WACAS)*. 2014.

[58] Sparsh Mittal. "A Survey of Techniques for Approximate Computing." English. In: *ACM Computing Surveys* 48.4 (2016), pp. 1–33. DOI: 10.1145/2893356.

[59] Hassan Ghasemzadeh Mohammadi, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "Efficient Statistical Parameter Selection for Nonlinear Modeling of Process/performance Variation." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.12 (2016), pp. 1995–2007. DOI: 10.1109/TCAD.2016.2547908.

[60] Debabrata Mohapatra, Vinay K. Chippa, Anand Raghunathan, and Kaushik Roy. "Design of Voltage-Scalable Meta-Functions for Approximate Computing." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011, pp. 950–955. DOI: 10.1109/DATE.2011.5763154.

[61] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[62] Vojtěch Mrázek, Radek Hrbacek, Zdeněk Vašíček, and Lukáš Sekanina. "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 258–261. DOI: 10.23919/DATE.2017.7926993.

[63] Ravi Nair. "Big Data Needs Approximate Computing: Technical Perspective." In: *Communications of the ACM* 58.1 (Dec. 2015), p. 104. DOI: 10.1145/2688072.

[64] Kumud Nepal, Soheil Hashemi, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. "Automated High-level Generation of Low-Power Approximate Computing Circuits." In: *IEEE Transactions on Emerging Topics in Computing* 7.1 (2019), pp. 18–30. DOI: 10.1109/TETC.2016.2598283.

[65] Kumud Nepal, Yueting Li, R. Iris Bahar, and Sherief Reda. "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6. DOI: 10.7873/DATE.2014.374.

[66] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0." In: *Journal on Satisfiability, Boolean Modeling and Computation* 9.1 (2014), pp. 53–58. DOI: 10.3233/sat190101.

[67] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[68] Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Adithya Parandhaman, and Anand Raghunathan. "Relax-and-retime: A Methodology for Energy-Efficient Recovery Based Design." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2013. DOI: 10.1145/2463209.2488871.

[69] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "ASLAN: Synthesis of Approximate Sequential Circuits." In: *Proceedings of the Design, Automation & Test in*

Europe Conference & Exhibition (DATE). IEEE, 2014. DOI: 10.7873/DATE.2014.377.

[70] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jörg Henkel. "Architectural-Space Exploration of Approximate Multipliers." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8. DOI: 10.1145/2966986.2967005.

[71] Ben Reynwar. *Decimation-in-Time Fast Fourier Transform*. [Online]. URL: https://github.com/benreynwar/fft-dit-fpga.

[72] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: Approximate Data Types for Safe and General Low-Power Computation." In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 164–174. DOI: 10.1145/1993498.1993518.

[73] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. "Approximate Storage in Solid-State Memories." In: *ACM Transactions on Computer Systems* 32.3 (2014), 9:1–9:23. DOI: 10.1145/2644808.

[74] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, and Laura Pozzi. "Partition and Propagate: An Error Derivation Algorithm for the Design of Approximate Circuits." In: *Proceedings of the Design Automation Conference (DAC)*. IEEE, 2019. DOI: 10.1145/3316781.3317878.

[75] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, Laura Pozzi, and Sherief Reda. "Approximate Logic Synthesis: A Survey." In: *Proceedings of the IEEE* 108.12 (99 2020), pp. 2195–2213. DOI: 10.1109/JPROC.2020.3014430.

[76] Ilaria Scarabottolo, Giovanni Ansaloni, and Laura Pozzi. "Circuit carving - A methodology for the design of approximate hardware." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 545–550. DOI: 10.23919/DATE.2018.8342067.

[77] Jeremy Schlachter, Vincent Camus, Christian Enz, and Krishna V. Palem. "Automatic Generation of Inexact Digital Circuits by Gate-Level Pruning." In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*. Lisbon. IEEE, 2015, pp. 173–176. DOI: 10.1109/ISCAS.2015.7168598.

[78] Jeremy Schlachter, Vincent Camus, Krishna V. Palem, and Christian C. Enz. "Design and Applications of Approximate Circuits by Gate-Level Pruning." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.5 (2017), pp. 1694–1702. DOI: 10.1109/TVLSI.2017.2657799.

[79] Deepashree Sengupta, Farhana Sharmin Snigdha, Jiang Hu, and Sachin S. Sapatnekar. "SABER: Selection of Approximate Bits for the Design of Error Tolerant Circuits." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2017, 72:1–72:6. DOI: 10.1145/3061639.3062314.

[80] Deepashree Sengupta, Farhana Sharmin Snigdha, Jiang Hu, and Sachin S. Sapatnekar. "An Analytical Approach for Error PMF Characterization in Approximate Circuits." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.1 (2019), pp. 70–83. DOI: 10.1109/TCAD.2018.2803626.

[81] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. "A Low Latency Generic Accuracy Configurable Adder." In: *Proceedings of the Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744778.

[82] Muhammad Shafique, Rehan Hafiz, Semeen Rehman, Walaa El-Harouni, and Jörg Henkel. "Invited - Cross-Layer Approximate Computing: From Logic to Architectures." en. In: *Proceedings of the Design Automation Conference (DAC)*. ACM Press, 2016. DOI: 10.1145/2897937.2906199.

[83] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. "Managing Performance vs. Accuracy Trade-Offs with Loop Perforation." In: *Proceedings of the Symposium and European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 124–134. DOI: 10.1145/2025113.2025133.

[84] F. S. Snigdha, D. Sengupta, J. Hu, and S. S. Sapatnekar. "Optimal Design of JPEG Hardware under the Approximate Computing Paradigm." In: *Proceedings of the Design Automation Conference (DAC)*. 2016, pp. 1–6. DOI: 10.1145/2897937.2898057.

[85] Mathias Soeken, Daniel Große, Arun Chandrasekharan, and Rolf Drechsler. "BDD minimization for approximate computing." In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 474–479. DOI: 10.1109/ASPDAC.2016.7428057.

[86] Phillip Stanley-Marbell et al. "Exploiting Errors for Efficiency: A Survey from Circuits to Applications." In: *ACM Computing Surveys* 53.3 (2020), 51:1–51:39. DOI: 10.1145/3394898.

[87] Robert Tibshirani. "Regression Shrinkage and Selection via the Lasso." In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. URL: http://www.jstor.org/stable/2346178.

[88] Shaghayegh Vahdat, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. "TOSAM: An Energy-Efficient Truncation- and Rounding-based Scalable Approximate Multiplier." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.5 (May 2019), pp. 1161–1173. DOI: 10.1109/TVLSI.2018.2890712.

[89] Zdeněk Vašíček. "Relaxed Equivalence Checking: A New Challenge in Logic Synthesis." In: *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2017. DOI: 10.1109/ddecs.2017.7968435.

[90] Zdeněk Vašíček. "Formal Methods for Exact Analysis of Approximate Circuits." In: *IEEE Access* 7 (2019), pp. 177309–177331. DOI: 10.1109/ACCESS.2019.2958605.

[91] S Venkataramani, A Sabne, V Kozhikkottu, K Roy, and A Raghunathan. "SALSA: Systematic Logic Synthesis of Approximate Circuits." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2012, pp. 796–801. DOI: 10.1145/2228360.2228504.

[92] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. "Quality Programmable Vector Processors for Approximate Computing." In: *Proceedings of the International Symposium on Microarchitecture (MICRO)*. ACM, 2013. DOI: 10.1145/2540708.2540710.

[93] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. "AxNN: Energy-Efficient Neuromorphic Systems Using Approximate Computing." In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED)*. ACM, 2014, pp. 27–32. DOI: 10.1145/2627369.2627613.

[94] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ACM, 2013, pp. 1367–1372. DOI: 10.7873/DATE.2013.280.

[95] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. "MACACO: Modeling and Analysis of Circuits for Approximate Computing." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 2011, pp. 667–673. DOI: 10.1109/ICCAD.2011.6105401.

[96] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity." In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[97] Tobias Wiersema. "Guaranteeing Properties of Reconfigurable Hardware Circuits with Proof-Carrying Hardware." PhD thesis. Paderborn, Germany: Paderborn University, 2021. DOI: 10.17619/UNIPB/1-1221.

[98] Tobias Wiersema and Marco Platzner. "Verifying Worst-Case Completion Times for Reconfigurable Hardware Modules using Proof-Carrying Hardware." In: *Proceedings of the International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE Computer Society, 2016. DOI: 10.1109/ReCoSoC.2016.7533910.

[99] Benjamin J. Winer. *Statistical Principles in Experimental Design*. New York: McGraw-Hill, 1971.

[100] Linus Witschen. "A Framework for the Synthesis of Approximate Circuits." Master's Thesis. Paderborn, Germany: Paderborn University, Aug. 2017.

[101] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." en. In: *Microelectronics Reliability* 99 (Aug. 2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

[102] Linus Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. "Jump Search: A Fast Technique for the Synthesis of Approximate Circuits." en. In: *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2019, pp. 153–158. DOI: 10.1145/3299874.3317998.

[103] Linus Witschen, Tobias Wiersema, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2018. URL: https://groups.uni-paderborn.de/agce/publications/pdfs/WitschenWMAP2018.pdf.

[104] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Making the Case for Proof-Carrying Approximate Circuits." Workshop on Approximate Computing (WAPCO). Workshop without proceedings. 2018. URL: https://groups.uni-paderborn.de/agce/publications/pdfs/WitschenWP2018.pdf.

[105] Linus Witschen, Tobias Wiersema, and Marco Platzner. "Proof-Carrying Approximate Circuits." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems. TVLSI* 28 (9 2020), pp. 2084–2088. DOI: 10.1109/TVLSI.2020.3008061.

[106] Linus Witschen, Tobias Wiersema, David Reuter, and Marco Platzner. "MUSCAT: MUS-based Circuit Approximation Technique." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022.

[107] Linus Witschen, Tobias Wiersema, Lucas David Reuter, and Marco Platzner. "Search Space Characterization for Approximate Logic Synthesis." In: *Proceedings of the Design Automation Conference (DAC)*. 2022. DOI: 10.1145/3489517.3530463.

[108] Linus Matthias Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. "Jump Search: A Fast Technique for the Synthesis of Approximate Circuits." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2019.

[109] Linus Matthias Witschen, Tobias Wiersema, and Marco Platzner. "Search Space Characterization for AxC Synthesis." Workshop on Approximate Computing (AxC). Workshop without proceedings. 2020.

[110] Clifford Wolf. *Yosys Open SYnthesis Suite*. [Online]. URL: http://www.clifford.at/yosys/.

[111] Yi Wu and Weikang Qian. "An Efficient Method for Multi-Level Approximate Logic Synthesis under Error Rate Constraint." In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2016. DOI: 10.1145/2897937.2897982.

[112] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate computing: A survey." In: *IEEE Design & Test* 33.1 (2016), pp. 8–22. DOI: 10.1109/MDAT.2015.2505723.

[113] Siyuan Xu and Benjamin Carrión Schäfer. "Exposing Approximate Computing Optimizations at Different Levels: From Behavioral to Gate-Level." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.11 (2017), pp. 3077–3088. DOI: 10.1109/TVLSI.2017.2735299.

[114] Makoto Yamada, Wittawat Jitkrittum, Leonid Sigal, Eric P Xing, and Masashi Sugiyama. "High-Dimensional Feature Selection by Feature-wise Kernelized Lasso." In: *Neural computation* 26.1 (2014), pp. 185–207.

[115] Amir Yazdanbakhsh et al. "Axilog: Language Support for Approximate Hardware Design." In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ACM, 2015, pp. 812–817. DOI: 10.7873/DATE.2015.0513.

[116] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. "On Reconfiguration-oriented Approximate Adder Design and its Application." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 48–54. DOI: 10.1109/ICCAD.2013.6691096.

[117] Georgios Zervakis, Konstantina Koliogeorgi, Dimitrios Anagnostos, Nikolaos Zompakis, and Kostas Siozios. "VADER: Voltage-driven Netlist Pruning for Cross-Layer Approximate Arithmetic Circuits." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.6 (2019), pp. 1460–1464. DOI: 10.1109/TVLSI.2019.2900160.

[118] Georgios Zervakis, Fotios Ntouskas, Sotirios Xydis, Dimitrios Soudris, and Kiamal Z. Pekmestzi. "VOSsim: A Framework for Enabling Fast Voltage Overscaling Simulation for Approximate Computing Circuits." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.6 (2018), pp. 1204–1208. DOI: 10.1109/TVLSI.2018.2803202.

[119] Georgios Zervakis, Sotirios Xydis, Dimitrios Soudris, and Kiamal Z. Pekmestzi. "Multi-Level Approximate Accelerator Synthesis under Voltage Island Constraints." In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 66-II.4 (2019), pp. 607–611. DOI: 10.1109/TCSII.2018.2869025.

[120] Liang Zhang, Mukul R. Prasad, and Michael S. Hsiao. "Incremental Deductive & Inductive Reasoning for SAT-based Bounded Model Checking." In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2004, pp. 502–509. DOI: 10.1109/ICCAD.2004.1382630.

[121] Ziji Zhang, Yajuan He, Jin He, Xilin Yi, Qiang Li, and Bo Zhang. "Optimal Slope Ranking: An Approximate Computing Approach for Circuit Pruning." In: IEEE, 2018. DOI: 10.1109/ISCAS.2018.8351238.

[122] Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Sanbao Su, Chang Meng, and Weikang Qian. "DALS: Delay-driven Approximate Logic Synthesis." English. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM Press, 2018. DOI: 10.1145/3240765.3240790.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available at:

https://bitbucket.org/amiede/classicthesis/

Final Version as of December 28, 2022 (1.0).