EPFL

Thèse n° 10 515

Contemporary Logic Synthesis: with an Application to AQFP Circuit Optimization

Présentée le 4 juin 2024

Faculté informatique et communications
Laboratoire des systèmes intégrés (IC/STI)
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Siang-Yun LEE

Acceptée sur proposition du jury

École
polytechnique
fédérale
de Lausanne

2024

The pursuit of knowledge and understanding is a lifelong journey.
— Claude Shannon

Acknowledgements

The PhD is a long learning road. At school, teachers may have been obligated to teach us knowledge because they get a salary explicitly for this. (Not saying we shouldn't be thankful to them, though!) But along the PhD journey, there have been so many people who helped us, directly or implicitly, without having an obligation to do so.

I would like to start by thanking my advisor, Prof. Giovanni (Nanni) De Micheli, who gave me great freedom and support, allowing me to freely try whatever I found interesting without having to worry about resources. I would also like to thank my jury committee, who spent the time and effort to read my thesis, traveled to attend my defense, and gave valuable feedback. Moreover, I want to thank Dr. Heinz Riener and Dr. Alan Mishchenko, who are not officially my advisors but have taught me a lot through our continuous collaboration.

A famous Taiwanese author wrote, *"For everything in life, we've got so much from others but used so little effort of ourselves; there are too many people to thank, so I would rather simply be grateful to God (to the world)."* (陳之藩〈謝天〉) Indeed, it would be too long to exhaustively list everyone I should thank, and even such a list may still be incomplete. Hence, I would refrain from enumerating more names to thank, such that no one would be disappointed. Nevertheless, please accept my sincere appreciation to every one of you whom I have met during these 4.5 years, both in academic and personal life. I truly believe that each one of the tiny interactions integrates into who I am today. Thank you.

Last but not least, I would like to thank myself. The journey to a PhD degree was not always easy and joyful; there were times of stress and disappointment. Nevertheless, I never really gave up trusting myself, which has been hard for someone with imposter syndrome. I have learned to accept imperfection, to encourage myself, and to acknowledge what I have accomplished. Thus, I want to thank myself for having grown into a better person through the PhD journey.

Munich, 18 April 2024 Siang-Yun (Sonia) Lee

Abstract

Electronic devices play an irreplaceable role in our lives. With the tightening time to market, exploding demand for computing power, and continuous desire for smaller, faster, less energy-consuming, and lower-cost chips, computer-aided design for electronics, or *electronic design automation* (EDA), becomes not only inevitable but also critical in the semiconductor industry. Being responsible for the transformation and optimization of switching circuits at the level of logic gates, logic synthesis plays a central role in modern EDA flows and is key to bringing up the *quality of results* (QoR). For several decades, logic synthesis techniques have been developed based on the properties and needs of *complementary metal-oxide-semiconductor* (CMOS) digital circuits and according to the available computing power. Recently, new challenges as well as opportunities have appeared and influenced the research directions of logic synthesis.

The development of logic synthesis and the advancement of *very-large-scale integration* (VLSI) designs are both enablers and challengers of each other. The up-scaling of computing systems stresses logic synthesis algorithms for their scalability, efficiency, and QoR. Conversely, better computing systems make computationally intensive strategies in logic synthesis affordable. A major part of contemporary logic synthesis research lies in its synergy with the exponential scaling of VLSI systems. Moreover, emerging alternatives to CMOS-based technologies pose new problems to be solved in EDA and logic synthesis. As an example, *adiabatic quantum-flux parametron* (AQFP) is a promising superconducting electronic technology featuring ultra-low switching energy dissipation. However, it has unconventional path-balancing and fanout-branching constraints to be considered in EDA.

This thesis presents a collection of novel approaches, demonstrating various aspects of contemporary logic synthesis. In the first part, we focus on technology-independent logic optimization with an emphasis on scalability while pushing the limits on QoR. In the second part, we show how new problems in EDA for emerging technologies like AQFP are approached, as well as how techniques presented in the first part are applied in AQFP circuit optimization.

The main technical proposals of this thesis are as follows. First, the proposal of a simulation-guided logic synthesis paradigm (a) sets the tone of the thesis, emphasizing additional QoR improvements with manageable runtime overhead. Then, the presentation of a family of heuristic resynthesis algorithms (b) complements the high-effort peephole optimization framework. At a higher level, we demonstrate that a simple design space exploration strategy (c), which discovers good optimization sequences on the fly, outperforms human-designed flows. To fulfill the special constraints imposed by the AQFP technology, we study possible constraint relaxations and their tradeoffs (d) and propose an AQFP technology legalization flow (e). Finally, by combining the proposed high-effort optimization (a, b) and other existing optimization algorithms with AQFP legalization (e) in the design space exploration framework (c), we achieve a significant 44% improvement over the state-of-the-art in the problem of AQFP circuit optimization.

To sum up, this thesis presents the essence of contemporary logic synthesis with an application in AQFP circuit optimization as an example. Indeed, in present days, the major challenge in logic synthesis lies in finding a "good-enough" local optimal in the huge design space while maintaining reasonable efficiency, as well as inventing or re-designing novel methods to tackle unconventional constraints imposed by emerging technologies.

Keywords: Electronic design automation, logic synthesis, superconducting electronics, adiabatic quantum-flux parametron

Zusammenfassung

Elektronische Geräte spielen eine unersetzliche Rolle in unserem Leben. Angesichts der immer kürzer werdenden Markteinführungszeiten, der explodierenden Nachfrage nach Rechenleistung und des ständigen Wunsches nach kleineren, schnelleren, weniger Energie verbrauchenden und kostengünstigeren Chips ist computergestütztes Design für Elektronik oder Elektronische Entwurfsautomatisierung (electronic design automation, EDA) in der Halbleiterindustrie nicht nur unvermeidlich, sondern auch entscheidend. Da die Logiksynthese für die Umwandlung und Optimierung von Schaltkreisen auf der Ebene von Logikgattern verantwortlich ist, spielt sie eine zentrale Rolle in modernen EDA-Abläufen und ist der Schlüssel zur Verbesserung der Qualität der Ergebnisse. Seit mehreren Jahrzehnten werden Logiksynthesetechniken auf der Grundlage der Eigenschaften und Anforderungen komplementärer Metall-Oxid-Halbleiter (complementary metal-oxide-semiconductor, CMOS)-Digitalschaltungen und entsprechend der verfügbaren Rechenleistung entwickelt. In letzter Zeit sind neue Herausforderungen und Möglichkeiten entstanden, die die Forschungsrichtungen der Logiksynthese beeinflussen.

Die Entwicklung der Logiksynthese und die Weiterentwicklung von VLSI-Designs (very large scale integration) sind sowohl förderlich als auch herausfordernd für beide Seiten. Die Aufwärtsskalierung von Rechensystemen stellt die Algorithmen der Logiksynthese in den Mittelpunkt ihrer Skalierbarkeit, Effizienz und QoR. Umgekehrt machen bessere Rechnersysteme rechenintensive Strategien in der Logiksynthese erschwinglich. Ein großer Teil der aktuellen Logiksyntheseforschung liegt in der Synergie mit der exponentiellen Skalierung von VLSI-Systemen.

Darüber hinaus werfen aufkommende Alternativen zu CMOS-basierten Technologien neue Probleme auf, die in EDA und Logiksynthese gelöst werden müssen. Ein Beispiel dafür ist das adiabatische Quantenfluss-Parametron (adiabatic quantum-flux parametron, AQFP), eine vielversprechende supraleitende elektronische Technologie, die sich durch einen äußerst geringen Energieverlust beim Schalten auszeichnet. Sie hat jedoch unkonventionelle Pfadausgleichs- und Fanout-Verzweigungsbeschränkungen, die in der EDA berücksichtigt werden müssen.

Diese These präsentiert eine Sammlung neuartiger Ansätze, die verschiedene Aspekte der modernen Logiksynthese aufzeigen. Im ersten Teil konzentrieren wir uns auf die technologieunabhängige Logikoptimierung mit dem Schwerpunkt auf Skalierbarkeit, während wir die Grenzen der QoR verschieben. Im zweiten Teil zeigen wir, wie neue Probleme in der EDA für aufkommende Technologien wie AQFP angegangen werden und wie die im ersten

Teil vorgestellten Techniken in der AQFP-Schaltungsoptimierung angewendet werden. Die wichtigsten technischen Vorschläge dieser Arbeit sind wie folgt. Zunächst gibt der Vorschlag eines simulationsgeführten Logiksynthese-Paradigmas (a) den Ton der Arbeit an und betont zusätzliche QoR-Verbesserungen mit überschaubarem Laufzeit-Overhead. Dann wird eine Familie von heuristischen Resynthese-Algorithmen (b) vorgestellt, die das aufwändige Peephole-Optimierungsverfahren ergänzen. Auf einer höheren Ebene zeigen wir, dass eine einfache Strategie zur Erkundung des Entwurfsraums (c), die gute Optimierungssequenzen im laufenden Betrieb entdeckt, die von Menschen entworfenen Abläufe übertrifft. Um die speziellen Einschränkungen, die durch die AQFP-Technologie auferlegt werden, zu erfüllen, untersuchen wir mögliche Lockerungen von Einschränkungen und deren Kompromisse (d) und schlagen einen Legalisierungsfluss für die AQFP-Technologie vor (e). Durch die Kombination der vorgeschlagenen High-Effort-Optimierung (a, b) und anderer bestehender Optimierungsalgorithmen mit der AQFP-Legalisierung (e) im Rahmen der Entwurfsraumerforschung (c) erreichen wir schließlich eine signifikante Verbesserung von 44% gegenüber dem Stand der Technik bei der Optimierung von AQFP-Schaltungen.
Zusammenfassend wird in dieser Arbeit das Wesen der modernen Logiksynthese am Beispiel der AQFP-Schaltungsoptimierung dargestellt. Heutzutage besteht die größte Herausforderung in der Logiksynthese darin, in dem riesigen Entwurfsraum ein lokales Optimum zu finden, das “gut genug” ist und gleichzeitig eine angemessene Effizienz aufweist, sowie neue Methoden zu erfinden oder neu zu entwerfen, um unkonventionelle Einschränkungen durch neue Technologien zu bewältigen.

Stichwörter: Elektronische Entwurfsautomatisierung (EDA), Logiksynthese, supraleitende Elektronik, adiabatisches Quantenflussparametron (AQFP)

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

EDA	Electronic Design Automation
IC	Integrated Circuits
VLSI	Very-Large-Scale Integration
RTL	Register Transfer Level
CMOS	Complementary Metal-Oxide-Semiconductor
FPGA	Field Programmable Gate Array
BDD	Binary Decision Diagram
SAT	Boolean Satisfiability
ILP	Integer Linear Programming
CNF	Conjunctive Normal Form
SMT	Satisfiability Modulo Theory
MAJ	Majority
MUX	Multiplexer
DAG	Directed Acyclic Graph
AIG	AND-Inverter Graph
MIG	Majority-Inverter Graph
XAG	XOR-AND-Inverter Graph
XMG	XOR-Majority-Inverter Graph
MuxIG	Multiplexer-Inverter Graph
TFI	Transitive Fanin

TFO Transitive Fanout

PI Primary Input

PO Primary Output

MFFC Maximum Fanout-Free Cone

SDC Satisfiability Don't Care

ODC Observability Don't Care

LUT Look-Up Table

SOP Sum Of Products

CEC Combinational Equivalence Checking

ECO Engineering Change Order

CEX Counterexample

SoTA State of The Art

QoR Quality of Results

AUT Application Under Test

SCE Superconducting Electronics

QFP Quantum-Flux Parametron

AQFP Adiabatic Quantum-Flux Parametron

JJ Josephson Junction

SFQ Single-Flux Quantum

NDRO Nondestructive Readout

B/S Buffers and Splitters

ASAP As Soon As Possible

ALAP As Late As Possible

EDP Energy-Delay Product

1 Introduction

Electronic devices and computing services are widely used in everyday life in modern society. With the ever-advancing development of deep learning, artificial intelligence, and other data-driven applications, demand for more intensive and efficient computation gives rise to dramatic increases in the complexity and compactness of electronic circuits. As an example, Apple's latest M2 Ultra microprocessor (released in June 2023) has 134 billion transistors on a single chip [New23]. It is obviously impossible for human engineers to design digital circuits down to the lowest details by hand. The field of *electronic design automation* (EDA) was born in the 1950s to assist the design of *integrated circuits* (ICs) and to avoid redundant manual effort. Today, EDA tools are inseparable from any *very-large-scale integration* (VLSI) design flow.

Logic synthesis is one stage in EDA positioned after *register transfer level* (RTL) synthesis and before physical design. It refers to the translation from a functional specification of Boolean logic into a gate-level representation of digital circuits, as well as the optimization of gate-level logic networks. Logic synthesis involves transformation between different technology-independent logic representations, identifying and removing logical redundancies hidden in these representations, and finally mapping into a technology-compatible representation. Although logic synthesis was not the first stage to be automatized in the history of EDA development, it has become an essential step and an important means to push the limits on the area, delay, power, performance, and other costs of the fabricated chips.

Logic synthesis stems from the theoretical foundation of Boolean algebra and switching logic built by George Boole [Boo47] and Claude Shannon [Sha38]. Around the 1980s and 1990s, logic synthesis techniques evolved from optimizing two-level logic representations to dealing with multi-level logic networks. With the invention of *binary decision diagram* (BDD) [Ake78], data structures used in logic synthesis also migrated from symbolic representations to more efficient abstractions. Later in the 2000s, homogeneous logic networks consisting of only one type of gates became a popular choice, separating efficient technology-independent optimization and technology mapping [MB06]. The development of heuristic NP-complete-problem solving tools such as *Boolean satisfiability* (SAT) solvers and *integer linear programming* (ILP) solvers further accelerated modern logic synthesis algorithms.

The rapid scaling of computing systems today both poses challenges and provides opportunities for logic synthesis researchers and engineers. Growth in computational power enables more computationally intense approaches to be adopted in logic synthesis in order to achieve limit-pushing chip size and efficiency results demanded by the newer generations of processors. There is thus a trend in contemporary logic synthesis to revisit and explore seemingly naive strategies that would not have worked a few decades ago due to the limited computational power back then. Perhaps surprisingly, with some carefully crafted computational hacks and smartly designed heuristic guidance, simple ideas are often shown to become powerful algorithms providing another several percent improvements that all companies crave.

Moreover, the increasing scaling demand contributes to the emergence of new technologies and computing paradigms, which have distinct characteristics compared to the traditional *complementary metal-oxide-semiconductor* (CMOS) digital circuit family. With their special design constraints and circuit properties, EDA tools must be adapted to fulfill their needs. An important direction of modern logic synthesis lies in either modifying existing algorithms or developing novel methodologies for emerging technologies. For example, in the second part of this thesis, we discuss logic synthesis for *adiabatic quantum-flux parametron* (AQFP), a promising superconductive circuitry providing ultra-low energy consumption. AQFP circuits are different from CMOS in two ways: (1) AQFP circuits are based on *majority* (MAJ) gates, instead of NAND gates; (2) AQFP circuits impose two unconventional constraints, namely path balancing and fanout branching. The MAJ-based property makes it a better choice to optimize AQFP circuits using *majority-inverter graphs* (MIGs), and the two constraints must be satisfied by an additional buffer and splitter insertion step. With AQFP circuit optimization as an example application, we demonstrate key aspects of contemporary logic synthesis in this thesis.

1.1 Electronic Design Automation

Electronic design automation (EDA) is an industry and an academic field aiming at the *computer-aided design* (CAD) of electronic circuits. Modern EDA flows can be separated into several stages (Figure 1.1), including:

1. *High-level and behavioral synthesis*: From a high-level system specification, the first stage is to synthesize and optimize system architecture designs. This stage typically includes resource allocation, scheduling, data path optimization, etc.

2. *Architectural and RTL synthesis*: This stage is responsible for optimizing the architecture at the RTL (sequential) level, for example by retiming. Then, combinational parts are separated from the large sequential design and a gate-level implementation is chosen for each RTL component.

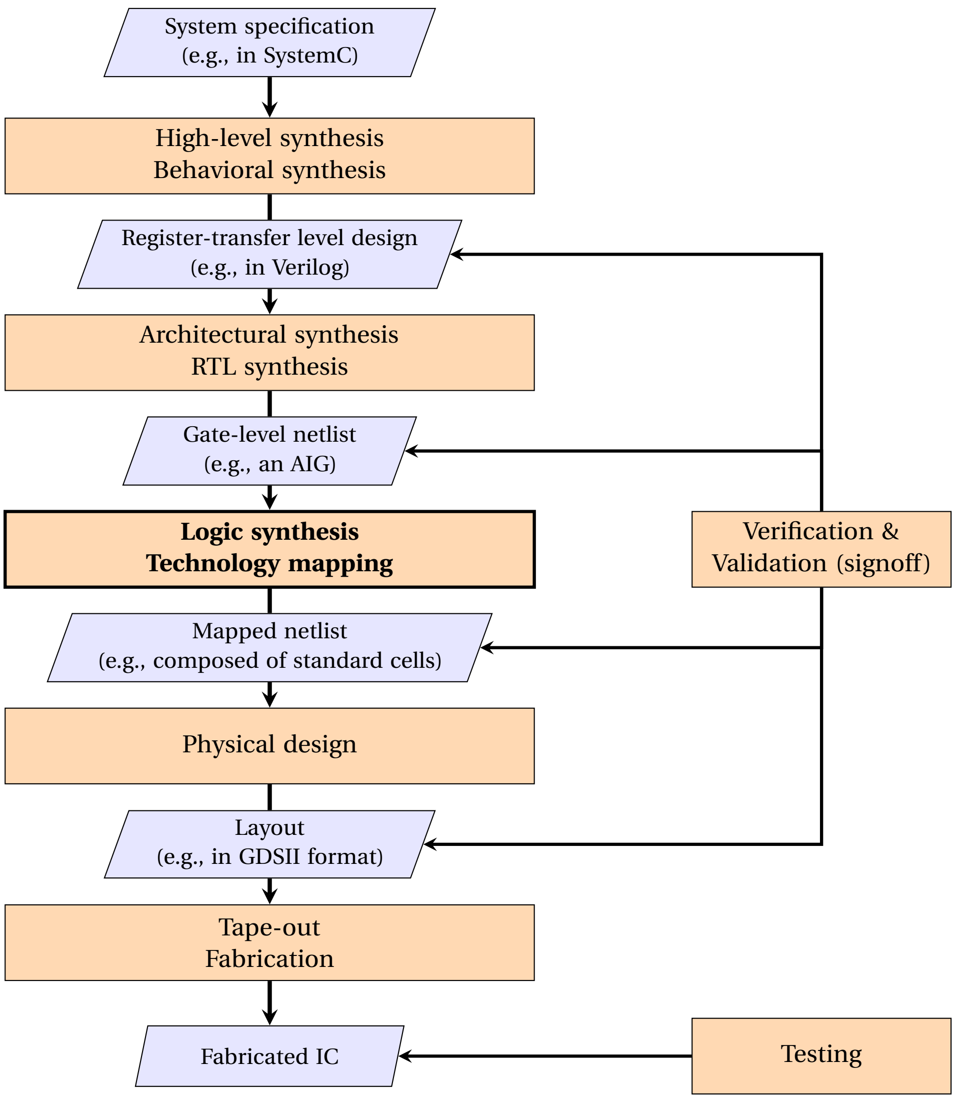


Figure 1.1: Simplified EDA flow and intermediate products.

3. *Logic synthesis and technology mapping*: Naive implementations of logic circuits usually contain lots of redundancies. In pursuit of more cost-efficient electronic systems, logic circuits are optimized at this stage to reduce their area and delay. In the end, generic logic representations are mapped into a technology-compliant circuit according to a given standard cell library of the specific technology.

4. *Physical design*: Physical design refers to the transformation from a gate-level netlist into a silicon layout ready for manufacture, including floor planning, placement, routing, clock tree synthesis, timing analysis, etc.

5. *Verification and validation*: After each stage, the produced circuit is compared against the original specification to ensure functional correctness. Moreover, timing and other design constraints are also verified.

6. *Testing*: Because there may be non-idealities in the manufacturing process, post-silicon testing must be done on the manufactured chips to identify possible defects.

Besides respecting the given specifications and constraints (i.e., correctness), EDA algorithms often emphasize their optimization aspect. Indeed, the hardest is usually not finding a legal and feasible solution, but one with the best *quality of results* (QoR). Various cost metrics are of concern in the IC industry, such as chip area, latency, throughput, energy consumption, density, wire length, and more. Moreover, there are often tradeoffs between these criteria. For example, with the same functionality, the *ripple-carry adder* takes a smaller area but has a higher latency, while the *carry-lookahead adder* achieves lower latency with the drawback of a bigger area [NIO96].

1.2 Logic Synthesis

In this thesis, the term *logic synthesis* refers to all steps after a gate-level netlist is obtained (from an RTL description, i.e., RTL synthesis) and before physical design can be performed. Logic synthesis flows typically consist of the following steps:

1. *Technology-independent logic optimization*: The input logic is broken down into a simple logic representation and optimized at the technology-independent level to minimize some given (often multiple and conflicting) cost metrics.
2. *Technology mapping*: The optimized representation is mapped to a target technology using a standard cell library.
3. *Post-mapping optimization and legalization*: The mapped circuit is further optimized with technology-specific optimization algorithms considering the constraints and parameters of the target technology.
4. *Verification*: To verify functional correctness, the optimized circuit is checked for logical equivalence against the original logic. Timing verification is also performed to avoid timing violations and glitches. Moreover, when there are special technology constraints to be fulfilled, an additional legality check is required.

Logic synthesis plays a central role in all EDA tools as the translation and optimization process from a functional specification to a structural description while meeting QoR goals in area, delay, and power.

Formulated by and named after George Boole in 1847, Boolean algebra [Boo47] provides the mathematical foundation for logic synthesis. A century later, in 1938, Claude Shannon introduced Boolean logic into the world of electrical computers, showing that Boolean algebra can be used to analyze switching circuits and proving that switching circuits designed in this way can compute anything Boolean algebra can solve [Sha38].

The problem of logic minimization was first tackled for two-level forms with an application in area optimization of *programmable logic arrays* (PLAs). As its name suggests, two-level forms, such as *sum of products* (SOP), represent Boolean logic functions using at most two layers of logical operations. The most famous two-level minimization algorithms are the Quine–McCluskey algorithm [Qui52; McC56] and Espresso [Bra+82]. These earliest logic synthesis algorithms still use symbolic representations or bitstreams as their underlying data structure for logic functions and their components.

However, in a VLSI design, digital circuits are multi-level netlists and two-level minimization is not enough. Thus, multi-level logic optimization gained increased interest since the 1980s with various pioneering academic tools being developed, including multi-level logic optimization systems MIS [Bra+87] and M32 [KS98], the Boulder optimal logic design system BOLD [Hac+89], the sequential logic synthesis system SIS [Sen+92], and the *field programmable gate array* (FPGA) logic synthesis system RASP [CPD96]. Simple, homogeneous graph representations are used in multi-level logic synthesis as the underlying data structure for logic circuits [BHS90], and auxiliary data structures like BDDs are involved in the core computations of logic manipulations and simplifications [YC02].

1.3 Trends in Logic Synthesis Techniques

Since the 2000s, a major portion of logic synthesis research has been developed around the *AND-Inverter Graph* (AIG) [Kue+02]. AIG is a technology-independent representation of multi-level logic networks involving only two-input AND gates and optionally-inverted wires. It gains popularity because its minimalistic data structure allows simple and efficient programs to be developed [BM06]. AIGs were first used in formal verification to simplify the equivalence-checking problem, but their application soon extended into technology-independent logic optimization. A notable academic logic synthesis and verification tool relying heavily on AIGs is ABC [BM10].

One of the core problems to be solved in logic synthesis, or logic optimization to be more specific, is identifying and removing logic redundancies to simplify the circuit. These redundancies are rooted in the flexibilities in logic representations, called *don't cares* [Bra83; DM93]. The numerous logic optimization methods existing in the literature [BHS90; De 94] can be roughly classified into two classes, namely *algebraic methods*, which treat Boolean functions as polynomials and optimize the logic network locally, and *Boolean methods*, which consider global and local Boolean logic and don't-care conditions to improve the optimization quality. Algebraic methods were popular in the earlier days because don't-care computation was based on BDDs which are not scalable. Since efficient don't-care computation techniques using truth tables and bit-parallel circuit simulation were proposed, Boolean methods have become mainstream.

The recent trend in logic synthesis in the twenty-first century favors efficient local transformations because they are more scalable. No matter how big the network is, small sub-networks of usually less than a hundred nodes are extracted and optimized independently. Such strategy is referred to as *peephole optimization* in this thesis. This way, the total runtime of an optimization algorithm can be linear to the size of the network. As an inevitable compromise, these algorithms are heuristics from a global optimization perspective. Indeed, the problem of logic optimization is intractable and there is no scalable and optimal approach. Nevertheless, with a portfolio of good heuristics, modern logic synthesis has been shown to reach near-optimal results in some cases.

1.4 Challenges and Opportunities of Contemporary Logic Synthesis

The drastic growth in the scale of digital circuits poses two types of challenges in logic synthesis. On the one hand, the scalability and efficiency of logic synthesis algorithms become crucial requirements while the QoR cannot be sacrificed, even though the underlying problems are intractable. On the other hand, as Moore's law reaches its bottleneck, engineers and researchers explore emerging beyond-CMOS electronic technologies and novel computing paradigms, seeking potential breakthroughs in density, throughput, latency, and energy efficiency. These new models often possess different characteristics than the traditional CMOS-based circuits and sometimes impose unconventional constraints, such that existing logic synthesis algorithms developed for CMOS-based circuits must be adapted accordingly.

Nevertheless, the increase in the available computational power nowadays provides opportunities for novel strategies in contemporary logic synthesis that were not advantageous or possible twenty years ago. Enhancements in parallel computing and improvements in heuristic problem-solving tools such as SAT solvers make it possible to solve some NP-hard problems in logic synthesis efficiently. Increased *central processing unit* (CPU) speed and *random-access memory* (RAM) size also allow some brute-force-like algorithms to become practically useful. The development of EDA is both forced by and results in the rapid advancement of digital integrated circuits with stronger computing power, forming a positive feedback loop.

This thesis presents new paradigms, methodologies and algorithms within a broad overview of contemporary logic synthesis, demonstrating how the opportunities are leveraged to tackle the challenges. It serves as a snapshot portrait of the ongoing development of logic synthesis in the 2020s, four decades after the birth of EDA as an industry.

1.5 Thesis Organization

This thesis is separated into two parts. In the first part (Part I, Contemporary Logic Synthesis, Chapters 3-6), we illustrate how carefully-designed heuristic approaches form the basis of contemporary logic synthesis, starting from a simulation-guided paradigm (Chapter 3), which leverages fast circuit simulation to extend the search space while maintaining scalability. Then, as the core of peephole logic optimization, we study heuristic methods to solve the resynthesis problem (Chapter 4), pushing the limits of high-effort optimization. As modern logic synthesis flows usually consist of multiple algorithms and iterations, we investigate an on-the-fly design space exploration framework (Chapter 5) which is shown to be effective in advancing the state-of-the-art QoR. Finally, we adapt existing automated testing and debugging techniques for logic synthesis algorithms (Chapter 6) to help improve the robustness of logic synthesis applications with minimal human effort.

In the second part (Part II, AQFP Circuit Optimization, Chapters 7-10), we demonstrate how contemporary logic synthesis techniques are applied to emerging alternative technologies, taking the optimization of AQFP superconducting circuits as an example. After the technology and its special characteristics and constraints are introduced in Chapter 7, we must first carefully study the design principles of AQFP sequential circuits to correctly establish an abstraction model and constraint formulation that make sense in practice (Chapter 8). Since the AQFP technology imposes special constraints to be fulfilled before physical design, we propose a series of legalization and optimization algorithms to be performed as the last step of logic synthesis (Chapter 9). In the end, we combine everything together as an AQFP technology mapping flow (Chapter 10): Logic optimization and AQFP legalization (Chapter 9) are interleaved in the design space exploration framework (Chapter 5). Among the various logic optimization algorithms involved in the flow, MIG resubstitution consists of using the MAJ-based resynthesis algorithm (Section 4.6) in the simulation-guided paradigm (Chapter 3).

The main chapters of this thesis (i.e., except for this chapter serving as an overview, Chapters 2 and 7 giving necessary background knowledge for the two parts, and Chapter 11 as conclusions) are based on published works. In the following, we summarize the technical contributions of each main chapter.

1.5.1 Chapter 3: Simulation-Guided Paradigm

This chapter is adapted from [Lee+22] (© 2022 IEEE, reprinted with permission). The contributions of this work are:

1. Proposes a simulation-guided logic synthesis and verification paradigm, which pre-generates and reuses expressive simulation patterns to reduce the efforts needed in SAT-based verification.

2. Presents methods to generate expressive simulation patterns, which are integrated with a bit-packing technique.

3. Demonstrates the benefits of the proposed paradigm with improved resubstitution quality and reduced SAT calls in CEC.

4. Shows the reusability of the pre-generated patterns across different applications and with network modifications with experimental results.

1.5.2 Chapter 4: Heuristic Resynthesis

This chapter is adapted from [LM23] (© 2023 IEEE, reprinted with permission), which is an extension to and summary of two previous works [Rie+22; LRD21]. The contents of this chapter are important for the following reasons:

1. Peephole optimization is a commonly used approach in modern logic synthesis, which selects small portions of a network and optimizes them locally. As the optimization core of this strategy, we carefully define and study the problem of logic resynthesis.
2. Three heuristic resynthesis algorithms for different network types are proposed: AND-based resynthesis was first proposed in [Rie+22] and MAJ-based resynthesis was first proposed in [LRD21], whereas MUX-based resynthesis is new in [LM23]. They have better complexities compared to existing exact algorithms while compromising with little sacrifice in the QoR compared to optimal solutions.
3. With their high efficiency and unlimited problem size, heuristic resynthesis is the only practical candidate to serve as the core of high-effort peephole optimization. Our experimental results show that the proposed techniques enable additional size reduction on benchmarks that are already highly optimized.

1.5.3 Chapter 5: Design Space Exploration

This chapter is adapted from [LRD23]. In this chapter, we study a higher-level problem in logic synthesis: Design space exploration.

1. Design space exploration is the problem of finding a good sequence of individual algorithms to apply on a specific benchmark, such that the best possible QoR can be achieved. This is a complicated and difficult problem. We discuss various common approaches.
2. In contrast to sophisticated strategies, we propose a simple, on-the-fly method to explore the design space and discover a good flow without requiring a complex search-based or AI-driven optimization framework. Despite the simplicity, we show that on-the-fly design space exploration outperforms flows designed by human experts.
3. New best results for the problem of MIG size optimization are presented as evidence. The proposed method is also used later in Chapter 10 in the context of AQFP circuit optimization and, again, gives impressive results.

1.5.4 Chapter 6: Testing and Debugging Logic Synthesis Algorithms

This chapter is adapted from [LRD22a]. In this chapter, we introduce modern testing and debugging techniques and adapt them specifically to gate-level logic networks. The main contributions are:

1. Our fuzz tester repeatedly generates small- and intermediate-sized netlists to hunt for bugs. We provide systematic approaches to test on small circuit topologies in addition to purely random networks.
2. Our testcase minimizer guarantees to isolate a minimal failure-inducing core of a potentially lengthy bug report. It reduces testcases more efficiently by adopting specialized structural reduction rules for gate-level networks.
3. Our methods are agnostic of the network type and support different gate-level netlist formats. This is the first time that automated debugging techniques are available for logic representations other than AIGs. We demonstrate with a case study that testing with more compact representations like XAGs increases the possibility of capturing rare defects.
4. Our implementations are tightly integrated into *mockturtle*, which eliminates interfacing overheads and provides about 10× speed-up over using external testing and debugging solutions.

1.5.5 Chapter 8: Impact of Sequential Design on AQFP Technology Constraints

This chapter is adapted from [LAD23] (© 2023 IEEE, reprinted with permission). The contributions are three-fold:

1. We re-examine the formulation of AQFP technology constraints and propose possible relaxations on these constraints: phase alignment instead of path balancing, as well as leveraging flexibilities on combinational inputs' splitting capacity and phases. We also discuss a potential issue with clock skew and the trade-off of adopting relaxed constraints.
2. We implement the first buffer-insertion framework which considers detailed and realistic constraints and possible relaxations. The framework is parameterized for easy customization of constraint specification.
3. We investigate the influence of technology constraints on JJ count. Using the relaxed constraints, a large portion of buffers can be saved. This observation can help scale up AQFP circuits which were bottle-necked by too many buffers before.

1.5.6 Chapter 9: AQFP Technology Legalization by Buffer/Splitter Insertion

This chapter is adapted from [Lee+24] (© 2024 IEEE, reprinted with permission), which summarizes a scalable and flexible framework for AQFP technology legalization and optimization, based on two previous papers [LRD22b; CD23][1]. The problem is systematically solved with the proposal of the following algorithms:

1. A linear-time irredundant buffer insertion algorithm that is locally optimal subject to a given schedule. This algorithm links the buffer count with a schedule of the network, showing that the buffer insertion problem is a scheduling problem.
2. Depth-optimal scheduling algorithms. They serve as starting points to obtain a legal schedule first, which can be further optimized later. (This part is mainly contributed by Alessandro[1].)
3. Heuristic optimization algorithms to minimize buffer count globally from a given schedule. Two orthogonal algorithms are presented: chunked movement and retiming. (Retiming is contributed by Alessandro[1].)
4. An AQFP legalization flow combining the algorithms above, which consists of obtaining an initial schedule, inserting buffers using the irredundant insertion algorithm, and then (optionally) further optimizing by interleaving chunked movement and retiming.

1.5.7 Chapter 10: AQFP Logic Synthesis Toolbox

This chapter is adapted from [Lee+24] (© 2024 IEEE, reprinted with permission). Logic synthesis for AQFP consists of majority-based logic optimization and technology legalization. They may be tackled independently for a faster runtime, but interleaving them has the potential of achieving better QoR. In this chapter, we summarize various algorithms presented in this thesis to form an AQFP logic synthesis flow.

1. For logic optimization, we combine the heuristic MAJ-based resynthesis algorithm and the simulation-guided paradigm as a high-effort MIG resubstitution algorithm.
2. We leverage the design space exploration framework to interleave logic optimization and technology legalization. We discuss how this approach allows exploring the design space in two orthogonal axes of MIG optimization and buffer minimization.
3. Verification is also briefly discussed, including functional and constraint verification.
4. Finally, experimental results show a significant 44% improvement in the energy-delay product compared to the best-known AQFP synthesis results.

To summarize, this thesis presents recent advancements in various aspects of contemporary logic synthesis, including the simulation-guided paradigm, high-effort resynthesis, on-the-fly

[1] [Lee+24] is a collaboration work with Alessandro Tempia Calvino. For completeness reasons, contents based on his work [CD23] are still summarized, but kept short and clearly marked in this chapter.

design space exploration, and tailored technology legalization for the emerging AQFP technology with unconventional constraints. These results show two important directions of logic synthesis development: First, while scalability and efficiency are still important, approaches that give the extra few percent improvement within moderate runtime overhead are gaining interest. Second, with new technologies with different constraints emerge, specialized algorithms need to be developed or adapted to fulfill their needs. At the end of this thesis, we present a complete flow for AQFP circuit optimization to demonstrate these properties of contemporary logic synthesis.

2 Background

2.1 Mathematical Abstractions and Data Structures for Logic Circuits

2.1.1 Boolean Logic

A *Boolean variable* is a variable taking values in the *Boolean domain* $\mathbb{B} = \{0, 1\}$. The (n-dimensional) *Boolean space* $\mathbb{B}^n$ is an n-ary Cartesian power of the Boolean domain. An (n-input, single-output, completely-specified) *Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{B}$ of n Boolean variables. Multi-output Boolean functions can be seen as an ordered set of single-output functions. A *minterm* of a Boolean function is a value assignment to all the function's input variables.

A *Boolean relation* $\mathcal{R}$ is a binary relation over two Boolean spaces $\mathcal{R} \subseteq \mathbb{B}^n \times \mathbb{B}^m$, a *domain* ($\mathbb{B}^n$) and a *codomain* ($\mathbb{B}^m$). Boolean functions are special cases of Boolean relations. When describing Boolean functions as Boolean relations, an element in the domain is a minterm of the Boolean function.

Boolean functions can be classified into two types:

- *Completely-specified* Boolean functions are special cases of Boolean relations where the relations are *functional* (i.e., an element in the domain maps into one unique element in the codomain) and *total* (i.e., every element in the domain maps into an element in the codomain).
- *Incompletely-specified* Boolean functions are Boolean functions for which the output values under some minterms are not specified. In other words, for some minterm $\vec{b} \in \mathbb{B}^n$, the output value can be either 0 or 1. In terms of Boolean relations, we have both $(\vec{b}, 0) \in \mathcal{R}$ and $(\vec{b}, 1) \in \mathcal{R}$.

When not specified, functions in this thesis refer to completely-specified, single-output

Boolean functions.

There are several possible representations of Boolean functions, such as propositional formulas, Boolean chains [Knu11], binary decision diagrams [Ake78], and truth tables. We use conventional symbols for logic operators (listed in Table 2.1) when writing propositional formulas.

2.1.2 Truth Tables

The *truth table* $T[f]$ of a k-input Boolean function $f : \mathbb{B}^k \to \mathbb{B}$ is a bit-string $u = u_1 \cdots u_l$, i.e., a sequence of bits, of *length* $l = 2^k$. The bit $u_i \in \mathbb{B}$ at the i-th position ($0 \leq i < l$), denoted as $T[f]_i$, is equal to the output of f under the input assignment (i.e., minterm) $\vec{a} = (a_1, \ldots, a_k)$, where

$$2^{k-1} \cdot a_k + \ldots + 2^0 \cdot a_1 = i. \tag{2.1}$$

If $T[f]_i = f(\vec{a}) = 1$, $\vec{a}$ is said to be an *onset* minterm; otherwise, if $T[f]_i = f(\vec{a}) = 0$, $\vec{a}$ is said to be an *offset* minterm.

We use

$$\textsc{Ones}(f) = \sum_{i=0}^{l-1} T[f]_i \tag{2.2}$$

to denote the number of 1-bits in the truth table of f, which is also the number of onset minterms, or the size of the onset.

Truth tables are manipulated by carrying out the usual Boolean operations on all of their bits. Suppose that $u = u_1 \cdots u_l$ and $v = v_1 \cdots v_l$ are two truth tables of length l, and $\alpha : \mathbb{B} \to \mathbb{B}$ and $\beta : \mathbb{B}^2 \to \mathbb{B}$ are, respectively, unary and binary Boolean operations, then $\alpha(u) = \alpha(u_1) \cdots \alpha(u_l)$ and $\beta(u, v) = \beta(u_1, v_1) \cdots \beta(u_l, v_l)$. Such truth table manipulations can be highly-efficiently implemented with the bit-parallel operations supported by modern CPUs [CSG99]. The bits of the truth tables are split into buckets of 32- or 64-bit machine words and each bucket is processed in one machine instruction.

2.1.3 Logic Networks

Logic networks (or simply *networks*) are technology-independent representations of digital circuits. A logic network N is a *directed acyclic graph* (DAG) defined by a pair (V, E) of a set V of nodes and a set E of directed edges. The node set $V = I \cup O \cup G$ is disjointly composed of a set I of *primary inputs* (PIs), a set O of *primary outputs* (POs), and a set G of *(logic) gates*. Each PI has in-degree 0 and unbounded out-degree, whereas each PO has in-degree 1 and out-degree 0. The out-degree of each gate is unbounded and the in-degree is a fixed number depending on the type of the gate.

Each element (n_i, n_o) in the edge set $E \subseteq V \times V$ models a wire between node n_i and node n_o, where the information flows from n_i to n_o, i. e., n_i is an input of n_o. n_i is said to be a *fanin* of n_o and n_o is said to be a *fanout* of n_i. The set of fanins of a node n is denoted by $\mathrm{FI}(n)$ and the set of fanouts of n is denoted by $\mathrm{FO}(n)$. Two nodes having a common fanout n_o (i.e., the fanin nodes of n_o) are said to be *siblings* of each other. In many practical network data structures, inverters are embedded on the edges. In other words, the edge set is extended to $E \subseteq V \times V \times \mathbb{B}$, where an element $(n_i, n_o, c) \in E$ models a wire from node n_i to node n_o with a complementation tag $c \in \{0 = \text{regular}, 1 = \text{complemented}\}$ recording the absence or existence of an inverter on the wire.

A path p in a network is a finite sequence $n_0, \ldots, n_l$ of nodes such that $(n_i, n_{i+1}) \in E, \forall 0 \le i < l$. We use $n_0 \overset{p}{\rightsquigarrow} n_l$ to denote that there is a path p from n_0 to n_l. The *transitive fanin* (TFI) or the *transitive fanout* (TFO) of a node n is the set of nodes such that there is a path between n and these nodes in the direction of fanin or fanout, respectively. A logic gate computes a Boolean function of its fanins and passes the resulting output value to its fanouts.

The size of a network (denoted by $|N|$) is determined by its number of nodes, and the depth of a network (denoted by $d(N)$) is the length of the longest path from a PI to a PO. This abstraction models the combinational part of digital circuits. In practice, PIs of a logic network are often provided by the register outputs of the previous sequential stage and POs are connected to the register inputs of the next stage.

Cuts

A *cut* in a network, defined over a given set $R \subseteq V$ of *root* nodes, is a set C of nodes such that any path from a PI to a root includes a node in C. Let $\textsc{Cuts}(R)$ denote the set of all cuts for the set R,

$$C \in \textsc{Cuts}(R) \text{ if } \forall i \in I, r \in R, \forall p : i \overset{p}{\rightsquigarrow} r, \exists n \in C : n \in p. \tag{2.3}$$

When R contains only one node n, $\textsc{Cuts}(R)$ may be abbreviated as $\textsc{Cuts}(n)$ and is also referred to as a cut of n:

$$\textsc{Cuts}(n) \triangleq \textsc{Cuts}(\{n\}). \tag{2.4}$$

Nodes in a cut are also called *leaves*. A cut is said to be k-feasible if the number of leaves does not exceed k. A node n is said to be *supported* by a set C of nodes if C is a cut of n. Given any set R of roots, the identical set $C = R$ is always a cut by definition, thus such cut is said to be a *trivial cut*. Also, the set I of PIs is always a cut for any possible R.

Cones and Windows

The *logic cone* between a cut $C \in \textsc{Cuts}(n)$ and a node n, often also called a *TFI cone* of n if C is unimportant or clear from the context, is the set of all nodes on any path from a node in C to n. All nodes in the logic cone are supported by C.

Conversely, a *TFO cone* of a node n is the set of all nodes on any path from n to a PO. Practically, we are often interested in TFO cones limited to a certain *depth* d, where the paths are limited to a length of at most d in the definition above.

The *maximum fanout-free cone* (MFFC) [CD94b] of a node n is the set of nodes in the TFI cone of n that only contributes to n. Specifically, a node m is said to be in the MFFC of n if all paths from m to any PO pass through n. Identifying the MFFC is important because the MFFC of a node n is the sub-graph that will be removed when n is removed.

A *window* is a sub-graph constructed from a *root* node r and a *cut* $C \in \textsc{Cuts}(r)$ and is used to extract the local functionality and for local optimization. A window always includes the logic cone between C and r. Additionally, depending on the application, nodes outside of the TFI cone of r but supported by C can also be added to the window. A window can be viewed as a smaller network with C as the set of PIs and r as a PO. The number of nodes in a window is also called the *volume* of the window. In practice, windows with a higher volume-to-cut-size ratio often contain more redundancy and are of higher interest in logic optimization.

Node Functions

Each node n in a network computes a Boolean function $f_n : \mathbb{B}^{|I|} \to \mathbb{B}$ in terms of the PIs, called the node's *global function*. To express the global functions, a Boolean variable x_i is associated with each PI $i \in I$. Let $\vec{x} = (x_1, \ldots, x_{|I|})$ be the set of all PI variables. By definition, the function of a PI node $i \in I$ is $f_i(\vec{x}) = x_i$. Then, in topological order, the functions of all nodes in the network can be computed by composing the functions of a node's fanins with the function of the corresponding logic gate. Finally, the PO functions are computed by taking the function of a PO node and inverting if the PO is complemented. Two nodes in a network are said to be *functionally equivalent* if their global functions are logically equivalent; otherwise, they are *functionally non-equivalent*.

The function of a node may also be expressed in terms of a cut supporting it. Given a node n and a cut $C \in \textsc{Cuts}(n)$, the *local function* $f_n^C : \mathbb{B}^{|C|} \to \mathbb{B}$ is the Boolean function derived by associating a Boolean variable with each node in C and computing the local functions of each node in the logic cone between C and n in topological order. The global functions are a special case of local functions using the PI set I as the cut:

$$f_n \triangleq f_n^I. \tag{2.5}$$

Types of Logic Networks

Prominent examples of logic networks include *And-Inverter Graphs* (AIGs) [Kue+02], where each node represents a two-input AND gate, and *Majority-Inverter Graphs* (MIGs) [AGD16], where each node represents a three-input *majority* (MAJ) gate. The MAJ gate computes the majority function MAJ of its fanins [MTT61], i.e.,

$$\mathrm{MAJ}(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3). \tag{2.6}$$

Extending the gate library with XOR gates, the *Xor-And-Inverter Graph* (XAG) [HFS17] is a logic network where nodes can be either a two-input AND gate or a two-input XOR gate. Similarly, the Xor-Majority Graph (XMG) [Haa+17] is extended from MIGs, where nodes can be three-input MAJ gate or three-input XOR gate.

Another interesting type of network is the *Multiplexer-Inverter Graph* (MuxIG), where each node represents a 2-to-1 *multiplexer* (MUX) gate. The MUX gate has three non-symmetric inputs: the S-input as the selection ("if") signal, the T-input as the "then" signal, and the E-input as the "else" signal. The function computed by a MUX gate can be written as

$$\mathrm{MUX}(s, t, e) = (s \wedge t) \vee (\neg s \wedge e). \tag{2.7}$$

2.2 Computational Tools

2.2.1 Boolean Satisfiability Problem

Boolean optimization methods are often formulated as a *Boolean satisfiability* (SAT) problem and solved by a SAT solver [Tov84; MS00]. A SAT problem asks whether a Boolean formula, usually presented in a *conjunctive normal form* (CNF) as a conjunction of *clauses*, is satisfiable. That is, whether there exists a value assignment making the formula evaluate to true. If so, the solver returns a *satisfiable* (SAT) result along with a satisfying value assignment; otherwise, it concludes that the problem is *unsatisfiable* (UNSAT). Logic networks can be translated into CNF formulae using the Tseytin transformation [Tse83].

By using SAT in logic optimization, we benefit from its global consideration of the Boolean functions and hence better optimization quality. However, SAT is an NP-complete problem [Sch78]. Although many approaches have been proposed to solve SAT problems efficiently for EDA applications [MS00] and efficient SAT solvers have been developed, SAT-solving is still slower than algebraic and local-search methods in general. In practice, to avoid the program being stuck in a difficult SAT solve, a *timeout* can be set to limit the time spent in solving SAT; and/or a *conflict limit* can be set to restrict the effort made by the SAT solver.

2.2.2 Integer Linear Programming

Linear programming is a mathematical optimization problem where constraints and the objective are specified as linear relationships. The problem asks to find a feasible value assignment to its variables, which satisfies all the constraints while optimizing toward the objective (maximizing or minimizing its value). *Integer linear programming* (ILP) is a linear programming problem where all variables are integers. In an ILP problem, the optimization objective is a linear combination of a subset of its integer variables, and the constraints are linear inequalities over its variables. ILP is NP-complete.

Many optimization problems can be formulated as ILP problems, such as the scheduling problem in high-level synthesis. Being both NP-complete, an ILP-feasibility problem can be encoded as a SAT problem and vice versa. Although exact algorithms to solve an ILP problem have high runtime complexities, there exist many well-performing heuristic algorithms and open-source tools.

2.2.3 Satisfiability Modulo Theory

A *satisfiability modulo theory* (SMT) problem is a generalization of the Boolean satisfiability problem [Bie+09]. A SMT problem asks whether a mathematical formula, interpreted within a certain formal theory, is satisfiable. For example, a satisfiability modulo integer linear algebra problem may ask whether a set of linear inequalities over integer variables is satisfiable, which is equivalent to asking whether an ILP problem is feasible. As SAT is already NP-complete, SMT problems are often NP-hard, depending on what the underlying theory is. Nevertheless, heuristic SMT solvers can be efficient in solving SMT problems with various theories and have been used in a wide range of applications [MB08].

2.3 Components of Logic Synthesis

2.3.1 Structural Analysis

Structural Hashing

Structural hashing is a technique integrated into homogeneous network data structures of most modern logic synthesis packages. In a homogeneous network, every node represents the same logic gate. Thus, two nodes having the same fanins (including the polarities of complementation on the fanin edges) must compute the same function. In other words, *structurally equivalence* implies *functional equivalence*. A hash table is used to efficiently identify structurally equivalent nodes during network construction as well as transformations.

Reconvergence-Driven Cut Computation

Two paths in a network are *reconvergent* if they start at the same node v_0, end at the same node v_l, and contain, respectively, two different fanins of v_l. Reconvergence is essential for don't-care-based optimization [Rie+22]. Having more reconvergent paths in the window also helps increase the volume of the window while being limited to the same cut size. These observations motivate the computation of a *reconvergence-driven cut* for a given root node as the first step of window construction.

In [MB06], a reconvergence-driven cut for a root node r is computed as follows. The *expand* operation

$$\textsc{Expand}(C, n) = (C - \{n\}) \cup \mathrm{FI}(n) \tag{2.8}$$

replaces a node n in a cut C with its fanins. The *cost*

$$\Delta(C, n) = |\textsc{Expand}(C, n)| - |C| \tag{2.9}$$

of expanding C on a node n is the difference in the cut size after and before expansion. If $\Delta(C, n) \leq 0$, we say it is a *cost-free* expansion. It is easy to observe that $\textsc{Expand}(C, n)$ is cost-free if and only if at most one fanin of n is not in C, i.e., iff $|\mathrm{FI}(n) - C| \leq 1$. Starting from the trivial cut $C = \{r\}$, the algorithm iteratively expands on the lowest-cost node in C, until the upper bound on cut size k is reached and there are no more cost-free expansions possible.

MFFC Computation

If the network data structure keeps a *reference counter* for each node that counts how many fanouts it has, then MFFC computation can be done efficiently by recursively dereferencing (decreasing the reference counter of a node, and recursively decreasing its fanins' reference counters if the reference count becomes zero) and then referencing (restore the reference counters) the root node.

Circuit Simulation

A *simulation pattern* (or abbreviated as a *pattern*) is a collection of Boolean values assigned to each primary input of a network. Circuit simulation is done by visiting nodes in topological order and computing their output values with their input values. In practice, several simulation patterns can be bundled together by using machine words, instead of a single bit, to represent a sequence of Boolean values. This way, 32 or 64 patterns can be computed for a node within a single CPU instruction using bitwise logical operations supported by modern arithmetic logic units. The *simulation signature* of a node is an ordered set of values produced at the node under each simulation pattern.

A set of simulation patterns is *exhaustive* if it covers all possible combinations of value assignment, which requires 2^k patterns for k PIs. The simulation signatures produced by simulating an exhaustive pattern set are also called *truth tables* and they completely specify the Boolean functions of the nodes.

Simulation can be done globally in the entire network or locally in a small window. In the former case, the simulation pattern set is possibly non-exhaustive because 2^{16} patterns are already impractical to handle, but the number of PIs is usually larger than 16. To use an exhaustive set of patterns, simulation must be restricted to a window of less than 16 (typically 8 to 10) leaf nodes.

2.3.2 Don't-Care Conditions

A *don't care* for an incompletely-specified function is a minterm for which the output value is not specified. The *don't-care set* of a function is the set of all of its don't care minterms. In a logic network, although all node functions (in terms of any cut) are completely specified, for some nodes, there may be some minterms where the output values of their functions are *flexible*. In other words, the function f_n^C of a node n in terms of cut C may be modified by changing its output value under some minterms without affecting the global functions of any PO. As a consequence, an incompletely-specified function where these minterms are don't cares and the output values under the other minterms are the same as f_n^C can be used to re-synthesize the logic cone between C and n. Two types of internal don't cares, arising from different reasons, may appear in logic networks:

Satisfiability don't cares

Given a cut $C \in \textsc{Cuts}(R)$ supporting a set R of nodes[1] and let $\vec{x} = (x_1, \dots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\vec{b}_C \in \mathbb{B}^{|C|}$ to $\vec{x}$ (i.e., a minterm of the local functions f_n^C of any node $n \in R$) is a *satisfiability don't care* (SDC) if this value combination never appears under any PI value assignment:

$$\nexists \vec{b}_I \in \mathbb{B}^{|I|}, (f_n(\vec{b}_I) : n \in C) = \vec{b}_C. \tag{2.10}$$

For example, an AND gate g_1 and an OR gate g_2 sharing the same fanins can never have $g_1 = 1$ and $g_2 = 0$ at the same time. This combination is a satisfiability don't-care of any node in the common TFO cone of g_1 and g_2.

[1]The supported set R is not involved in the definition of SDCs, so it can, in theory, be empty and C is not necessarily a cut. Although one may define and compute SDCs for any set C of nodes, in practice, SDCs are only meaningful when C is indeed a cut, as SDCs are used to optimize nodes in R.

Observability don't cares

Given a node n and a cut $C \in \textsc{Cuts}(n)$ and let $\vec{x} = (x_1, \ldots, x_{|C|})$ be Boolean variables associated with each node in C, a value assignment $\vec{b}_C \in \mathbb{B}^{|C|}$ to $\vec{x}$ (i.e., a minterm of the local function f_n^C) is an *observability don't care* (ODC) with respect to n if none of the PO functions are affected by flipping the output value of f_n^C under $\vec{b}_C$:

$$\forall \vec{b}_I \in \mathbb{B}^{|I|}, (f_n(\vec{b}_I) : n \in C) = \vec{b}_C \implies \forall o \in O, f_o^*(\vec{b}_I) = f_o(\vec{b}_I), \tag{2.11}$$

where f_o^* is the PO function derived by replacing any regular outgoing edge of n with a complemented one and replacing any complemented outgoing edge of n with a regular one. The value assignment $\vec{b}_C$ is said to be *unobservable* with respect to n.

Computation of Internal Don't Cares

The appearance of "don't care" as a technical term in the literature dates back to as early as the 80s [Bra83]. Pioneering research attempted to derive don't care in multi-level networks and use them in two-level minimization to resynthesize part of the network [Bar+88]. Theories on don't-care computation were formulated based on symbolic computations propagated through the network [Mur+89; DM93]. Until the late 90s, computation of don't cares had been implemented using *binary decision diagrams* (BDDs). Due to scalability concerns, approximated computation was adopted [MB90], and the compatibility of ODCs was studied to avoid re-computation of ODCs in the network once an ODC is used to change the function of a node [SK04]. Since the early 00s, computation tools of don't cares have moved from BDDs to SAT, enabling using complete, instead of approximate, don't cares while maintaining scalability [MB05].

In many modern logic synthesis tools, internal don't cares are derived locally (under-approximated) using bit-parallel circuit simulation:

- To compute the SDCs for a given set C of nodes, we first find another cut $C_0 \in \textsc{Cuts}(C)$ supporting C. Then, we perform circuit simulation by assigning projection functions to nodes in C_0 and obtain the local functions of nodes in C in terms of C_0, represented as truth tables. Finally, by analyzing each bit in the truth tables, we identify the value combinations at C that do not happen, which are the SDCs at C.

- To compute the ODCs with respect to a node n, we first mark the TFO cone of n for a predefined depth and collect the set R of nodes having fanouts outside of this transitive fanout cone. Then, we find a cut $C \in \textsc{Cuts}(R)$ supporting R and perform circuit simulation to obtain the local functions f_R of nodes in R in terms of C. After adding a temporary inverter at the output of n, we perform another simulation to obtain f_R^*. Finally, we compare the two simulation results to identify the minterms where f_R and f_R^* have identical values, which are the ODCs with respect to n.

2.3.3 Exact Synthesis and Databases

Although the problem of finding the smallest circuit implementing a given logic function is intractable, it can still be solved for smaller functions with about less than 10 input variables. Such algorithms are called *exact synthesis*. Since exact synthesis algorithms have high runtime complexity, small optimum circuits are often pre-generated and saved in a database, so that they can be retrieved quickly during logic optimization. Such databases usually contain one or more implementations of all functions or a subset of practical functions up to a certain number of inputs. *NPN classification* is often used to reduce the number of entries because functions in the same NPN class (i.e., functions differ by input negation, input permutation, and/or output negation) may share the same optimum circuit.

There are two main approaches for database generation: SAT and enumeration. SAT-based exact synthesis encodes the question "Does a network with r gates that implements function f exist?" as a CNF formula and uses a SAT solver to find a feasible solution. In the formulation, Boolean variables are used to encode the interconnections between gates and the functions of each gate. The number of gates r is a fixed assumed value, which also affects the number of variables and clauses in the formula. To find the smallest network, we start from a smaller value of r, increase it if the formula is UNSAT, and solve iteratively until a feasible solution is found. An optimum database can also be generated by enumerating all possible circuits and recording the smallest ones seen for each NPN class [Lee+19].

2.4 Modern Logic Synthesis Algorithms

2.4.1 Algebraic and Boolean Methods

Algebraic Methods: Polynomial-Algebra-Based Optimization

Algebraic methods are one of the earliest approaches to optimizing logic circuits. These methods represent Boolean functions symbolically and treat them as polynomial expressions, for example in the SOP form. Inspired by polynomial algebra, operations like division, substitution, and common sub-expression extraction are developed for Boolean functions written in the polynomial form. These operations form the basis for decomposition and simplification algorithms such as kernel extraction, factorization [Bra82], balancing [Mis+11a], refactoring [Haa+18], and algebraic rewriting [YCM17]. Although simple and fast, algebraic methods neglect the Boolean nature of logic functions and thus miss some optimization opportunities.

Boolean Methods: Don't-Care-Based Optimization

In contrast to algebraic methods, Boolean methods often achieve better optimization quality because they consider the flexibilities of the network. In other words, they incorporate some form of don't-care computation and utilize don't-care conditions to find more optimization opportunities. Modern logic optimization algorithms are mostly don't-care-based, and the key to scalable implementations is efficient don't-care computation. The remainder of this section introduces some common Boolean methods.

2.4.2 Cut Rewriting

Cut rewriting (or simply *rewriting*) is a peephole optimization algorithm that leverages a database of small optimum circuits. It works by enumerating k-feasible cuts (where k is usually 4) for each node in the network, simulating each cone to obtain the local functions, looking up in the database, choosing the best cut that gives the most gain, and replacing the chosen cone with the optimum implementation in the database [MB06]. The algorithm is said to be DAG-aware because when evaluating the potential gain of each replacement, it is aware of the fact that the network is a structurally hashed DAG and that structurally equivalent nodes can be shared [MCB06; RMS20; Rie+19a].

2.4.3 Boolean Resubstitution

Boolean resubstitution (or simply *resubstitution*) aims at reducing the size of a logic network by trying to resynthesize each node using existing nodes in the network. For each node in a network, called the *root*, the algorithm tries to find a smaller replacement for the MFFC of the root. If the root node is replaced and deleted, all nodes in its MFFC can also be deleted, reducing the size of the network. Resubstitution is also classified as a peephole optimization algorithm.

The replacement for the root node, called the *dependency circuit*, is built upon a set of potentially useful nodes existing in the network, called *divisors*. A divisor should not be in the TFO cone of the root, otherwise the resulting network would be cyclic. It should also not be in the MFFC because nodes in the MFFC are to be removed after resubstitution. Nodes depending on primary inputs that are not in the TFI of the root node can also be filtered out from the set of divisors because their functions are unrelated to that of the root node. In practice, to keep the runtime reasonable, windows constructed with k-feasible cuts are often used to collect the divisors.

A *resubstitution candidate* (also abbreviated as a *candidate*) is either a divisor itself or a single-output function, named the *dependency function*, built with several divisors. In the latter case, the candidate is represented by the top-most node of the dependency circuit. A *resubstitution*, or simply *substitution*, is a pair (r, c) of a root node r and a resubstitution candidate c, and it is said to be *legal* if replacing r with c does not change the functions of any PO. Otherwise, the resubstitution is said to be *illegal*.

Research in Boolean resubstitution techniques dates back to the 1990s [Sat+91; KS98]. In the 2000s, efforts were made to improve the scalability of BDD-based computations [KK04] and to move away from BDDs to simulation and SAT solving [Mis+06b; Mis+11b]. In [Mis+06b], the dependency function is computed by enumerating its onset and offset cubes using SAT and interpolation [Cra57], where random simulation is used for the initial filtering of potentially useful divisors. In [Mis+11b], structural analysis (windowing) was introduced to speed up the algorithm further. Windowing is used to limit the search space and the SAT instance size, with the inner window as a working space, and the outer window as the scope for computing don't-cares.

An efficient Boolean resubstitution algorithm for AIGs using windowing was presented in [MB06]. It relies entirely on truth table computation, without any use of BDDs or SAT. The search for divisors is limited to a window near the root node, which is constructed from a size-limited cut to allow exhaustive simulation. The node functions in the window are expressed in terms of the cut nodes. The dependency function is not computed as a separate step after minimizing its support, as in [Mis+11b]. Instead, simple dependency circuits of up to three AND gates are explicitly tried for resubstitution using several heuristic filters. This windowing-based and truth-table-based resubstitution framework has been generalized for many different gate types including majority gates [Rie+18] and complex gates [Ama+18].

2.4.4 Technology Mapping

After technology-independent logic optimization is performed on homogeneous network data structures, *technology mapping* is required as the last step in logic synthesis to transform the network into one that is compatible with the underlying technology for fabrication. For example, for an *application-specific integrated circuit* (ASIC), logic gates must be chosen from a library where a transistor-level circuit design for each gate is available. In contrast, for FPGA synthesis, a graph consisting of *look-up tables* (LUTs) with no more than a certain number (k) of inputs, called a *k-LUT network*, is needed. The latter case requires a *LUT mapping* algorithm, which is a special case of technology mapping [CD94a; CD94b; MCB07; CM10; Ray+12].

A technology mapping algorithm maps from a *subject graph*, which is the input to the algorithm, into a *mapped graph*, which is the output of the algorithm. In [Tem+22], a versatile mapper was developed, which is capable of mapping from any subject graph into a mapped graph using any library. Cuts are often enumerated in technology mapping to compute local functions in the subject graph, which are then used to choose suitable gates from the library to substitute the cuts in the mapped graph. Along the mapping process, optimizations on area or delay can be performed. For LUT mapping, it has been shown that optimal delay can be achieved in linear time [CD94a].

2.4.5 Combinational Equivalence Checking

Combinational equivalence checking (CEC) is the problem asking whether two (combinational) logic networks are functionally equivalent. A *miter* network is often constructed in the process

Table 2.1: Symbols and special functions used in this thesis.

Symbol	Meaning
$\neg$	Logic operator NOT
$\wedge$	Logic operator AND
$\vee$	Logic operator OR
$\oplus$	Logic operator XOR
$\leftrightarrow$	Logic operator XNOR
MAJ$(\cdot)$	Majority-vote function
MUX$(\cdot)$	2-to-1 multiplexing function
FI$(\cdot)$	Set of fanins of a node
FO$(\cdot)$	Set of fanouts of a node
ONES$(\cdot)$	The number of 1-bits in the truth table of a function
CUTS$(\cdot)$	The set of cuts of a node or a set of nodes
$d(\cdot)$	The depth of a network

of solving a CEC problem. A miter of two networks N_1 and N_2 having the same number of PIs and the same number of POs is a network N_m consisting of N_1 and N_2, where the corresponding PIs in N_1 and N_2 are connected to the same PI of N_m, the corresponding POs of N_1 and N_2 are pair-wisely fed into XOR gates, and the XOR gates are then fed into one OR gate, whose output is the only PO of N_m. The miter network outputs 1 if and only if there exists an input assignment such that N_1 and N_2 compute different values in at least one PO. The CEC problem is equivalent to asking whether the miter of two networks produces a constant-0 output.

When the number of PIs is small, CEC can be solved by exhaustive simulation of the miter network. Otherwise, SAT solvers are often used to formally prove that the miter produces constant-0 output by asking whether there exists an input assignment such that the miter output is 1. To facilitate SAT solving and reduce runtime, the miter network can be optimized first to reduce its size. This is why CEC is often studied together with other logic optimization problems.

In [Mis+06a], further improvements to CEC are proposed. Instead of proving the entire miter network, which is often big, the authors propose to use random simulation to identify potential equivalent nodes and leverage them as stepping stones.

2.5 List of Symbols and Common Variables

Symbols for logical operations and some special functions used in this thesis are listed in Table 2.1. Also, certain variables have a fixed meaning throughout the thesis and are listed in Table 2.2.

Table 2.2: Variables with a fixed meaning in this thesis.

Variable	Meaning
$\mathbb{B}$	Boolean domain
f	A Boolean function
n	A node
k	Cut size or number of variables
m	User-specified maximum size of dependency circuits in resubstitution and resynthesis algorithms
N	A logic network
I	Set of PIs
O	Set of POs
G	Set of divisors (Chapters 3 and 4) or set of gates (Part II)
H	Dependency circuit (Chapters 3 and 4)
B	Set of buffers (Part II)
$\mathcal{S}$	A schedule (Part II)

2.6 Summary

In this chapter, we introduced the foundation of contemporary logic synthesis, from mathematical abstractions and data structures to model Boolean logic to powerful computational tools and efficient algorithms for structural analysis and don't-care computation. These basic concepts are important to the scalability of logic synthesis systems nowadays. We also presented prominent examples of modern logic synthesis algorithms commonly used in academic as well as commercial tools. Based on these, the remainder of this thesis proposes novel frameworks and algorithms to further advance the efficiency and QoR of logic synthesis flows.

Contemporary Logic Synthesis Part I

3 Simulation-Guided Paradigm

3.1 Motivation

As the size and complexity of digital circuits grow, there is often a trade-off between efficiency and quality. Algebraic methods, as well as other local search methods such as structural analysis and window simulation, are efficient but often sacrifice optimality. In contrast, Boolean methods usually achieve better quality at the cost of solving NP-hard Boolean problems using a BDD package in earlier research or a SAT solver in more recent literature.

To balance between the two extremes, circuit simulation is often used in Boolean methods as an efficient approximator of the Boolean functions embedded in logic networks. In functional reduction [Mis+05], random and guided simulations are used to identify equivalent nodes and merge them. In combinational equivalence checking [Mis+06a], simulation is also used to find cut-points between two networks that serve as stepping stones for the proof of equivalence at the primary outputs. However, if the simulation is not exhaustive, formal verification, which is usually done with SAT-solving, is still required [Mis+06b]. In [MB05; Mis+06b], a combination of random simulation and SAT solving was proposed to compute flexibilities (don't-cares) of Boolean networks within a window and to compute the dependency function in resubstitution.

Motivated by the efficacy of these techniques adopting *random* simulation, in this chapter, we introduce the *simulation-guided paradigm* for logic synthesis and verification, where efforts are made in pre-generating a set of high-quality, *expressive* simulation patterns to further strengthen the power of simulation. By increasing the expressive power of the simulation patterns, synthesis and verification algorithms become more efficient, and the extension of the search space in optimization algorithms becomes more affordable. The underlying hypothesis, which is confirmed by experimental results, is that expressive simulation patterns can be amassed for a logic network and used later as an efficient filter to avoid unnecessary SAT solver calls.

Moreover, these patterns can be reused multiple times to speed up logic synthesis and verification for the same or a similar network in various applications. Inspired by the success of *counter-example-guided abstraction refinement* (CEGAR) in the domain of software model checking [Cla+00], the simulation-guided paradigm also refines the pre-generated patterns throughout the process of logic synthesis with counter-examples generated by SAT solving.

The proposed paradigm is useful for algorithms dominated by expensive Boolean computations. Two representative applications are presented in this chapter: *Boolean resubstitution* (introduced in Section 2.4.3) and *combinational equivalence checking* (introduced in Section 2.4.5). We assume in this chapter that the underlying data structure for logic networks is AIG, as it is widely used in logic synthesis. Nevertheless, this paradigm can also be applied to other types of homogeneous logic networks, such as MIGs and XAGs, as well as mapped networks such as k-LUT networks [MCB07].

3.2 Overview

The simulation-guided paradigm is proposed and described in Section 3.3. As a core component of the paradigm, strategies to generate simulation patterns based on stuck-at-value testing [CR88] and observability checking [DD90], as well as a bit-packing technique to compress the generated patterns, are presented in Section 3.4.

In Section 3.5, the first representative application of the proposed paradigm, simulation-guided Boolean resubstitution, is demonstrated. The classic resubstitution algorithm iterates over the nodes in a logic network and attempts to re-express their functions using other nodes in the network. In simulation-guided resubstitution, nodes fed into the resynthesis engine are represented by their simulation signatures, and a SAT solver is used to validate the computed resubstitution candidates. Using expressive simulation patterns, most illegal candidates can be quickly identified and ruled out within the engine by simply comparing simulation signatures, without the need for SAT-based validation. Experimental results show that simulation-guided resubstitution allows user-specified tuning of the efficiency-quality trade-off and improves optimization quality by considering a larger search space while maintaining reasonable efficiency. Compared to a state-of-the-art AIG resubstitution algorithm [MB06], the average reduction in the number of AIG nodes improves from 3.65% to 5.90%.

In Section 3.6, the second representative application, simulation-guided equivalence checking, shows that expressive simulation patterns are also useful in verification. Similarly, simulation-guided CEC leverages the expressive patterns generated in earlier synthesis stages to disprove more non-equivalent nodes than random simulation can do, thus reducing the effort needed in SAT-based formal verification. In our experiment, a 9.5% reduction in the number of SAT calls is achieved when expressive patterns are used in CEC.

This motivates us to study what makes simulation patterns expressive and profile different pattern generation strategies, including random simulation, the proposed stuck-at-value-based and observability-based methods, and combinations of these. In Section 3.7, we test and compare the expressive power of various simulation pattern sets. In the process of resubstitution and CEC, pre-computed simulation patterns can be refined further with the counter-examples generated by SAT-solving. The generated patterns and the supplemented counter-examples can be reused in two schemes: across different algorithms, such as resubstitution followed by CEC, and across different versions of the same design. Reusability in the latter case is verified with experiments on *engineering change order* (ECO) [Jar+11] benchmarks, which are similar networks with functional modifications.

3.3 The Simulation-Guided Paradigm

This chapter introduces a new paradigm for logic synthesis and verification that exploits fast bit-parallel simulation to reduce the number of expensive NP-hard equivalence checks based on SAT. The rationale behind the idea is to pre-compute a set of simulation patterns for a given logic network, which can efficiently rule out most non-equivalences by simply comparing simulation signatures. Motivated by the fact that detecting and verifying functional equivalence are the major tasks in many logic optimization (especially Boolean methods) and verification algorithms, we define *expressive* simulation patterns as follows.

Definition 3.1. A non-exhaustive set of simulation patterns for a logic network is said to be *expressive* if the simulation signatures obtained by simulating the patterns can be used to pair-wisely distinguish functionally non-equivalent nodes that either already exist in the logic network or can be derived from some existing nodes. ■

The exhaustive set of simulation patterns satisfies the latter part of this definition, but this is typically too large for logic networks with 16 or more primary inputs. In practice, only expressive simulation patterns that can be efficiently stored and simulated using less than, say, a few hundred or thousand bits are of interest.

We assume that, for a given logic network of interest, a set of expressive simulation patterns with size proportional to the network size can be found. This means that the expressive simulation patterns can be pre-computed, stored, and reused by different logic synthesis or verification algorithms when applied to the same network, or by the same algorithm when invoked multiple times with slightly different networks. The assumption is verified with experimental results in Section 3.7 by showing pattern reusability after ECOs, which are typically small functional modifications to networks under design [Jar+11]. With this assumption, we claim that the time needed to generate the expressive patterns is not critical because they will be reused many times such that the benefits are more substantial.

Figure 3.1 illustrates the proposed simulation-guided paradigm. For each design (named `design1`), a set of expressive simulation patterns is generated once (`design1.pat`) and is used several times in the logic synthesis and verification flow. The same pattern set is also applicable for various versions of the design with functional modifications (`design1_v1`, `design1_v2`, etc.). When the pattern set is used in one of the simulation-guided algorithms, it

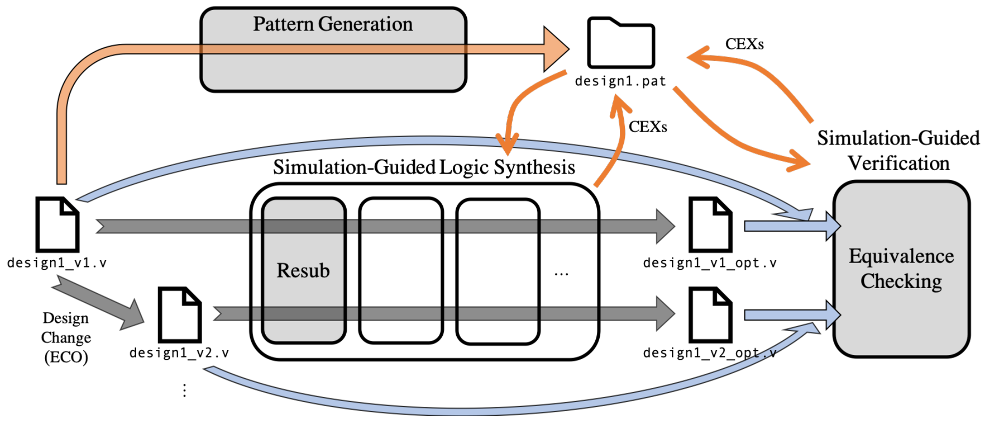


Figure 3.1: The simulation-guided logic synthesis and verification paradigm.

is supplemented and refined with the counter-examples (CEXs) generated as side-products during the execution of the algorithm. The blocks shaded in grey are implemented and described in this chapter. While other logic synthesis algorithms may also benefit from adopting the paradigm (the blank blocks in the figure), we present only resubstitution and CEC as examples in this chapter.

Expressive simulation patterns cannot be derived directly from the Boolean functions of the primary outputs, but must account for some structural information of the network. An intuitive explanation of this observation is that a PO function can be implemented by a large number of structurally different logic networks. Despite this, the idea of reusing simulation patterns in multiple optimization or verification runs is still valid because the initial structure of the network often is determined by high-level synthesis and later carefully fine-tuned by logic optimization. Consequently, only a small fraction of closely related structures are encountered during logic optimization and the final verification of the network.

The proposed simulation-guided paradigm can be adopted by algorithms dealing with the Boolean relation among nodes in logic networks. For example, in Boolean resubstitution, simulation signatures can be used as an approximation of node functions when finding resubstitution candidates. This way, restriction to local windows is avoided and global information is utilized at a low cost. As simulation patterns are already generated for the optimization algorithms prior to verification, reusing them in CEC comes at no extra cost. With their stronger ability to distinguish non-equivalent nodes without SAT solving, the overall number of SAT calls in CEC can be reduced. The paradigm is potentially suitable for other algorithms, such as the computation of structural choices [Cha+06], to improve the quality of mapping and gate matching between several versions of the same logic network. Furthermore, the resulting patterns can also be used in *automatic test pattern generation* (ATPG) [Rot66] and in circuit reliability analysis [CM10].

To conclude, simulation signatures are used as efficient approximations of node functions to reduce NP-hard equivalence checks. As they may not cover all circuit states under all possible input assignments, formal verification (in this chapter, by SAT-solving) is inevitable in simulation-guided algorithms. As byproducts, counter-examples in terms of PI value assignments, i.e., new simulation patterns, are generated. To reduce unnecessary SAT-solving, we seek to increase the accuracy of such approximation by partial simulation. On one hand, we propose to pre-generate an expressive pattern set to be reused across multiple optimization runs and across different algorithms, and we study methods to ensure the good quality of these patterns in the first place. On the other hand, motivated by the success of various counter-example-guided logic synthesis and verification works [Cla+00; AA20; Mis+05; Mis+06b], we propose to collect and keep the counter-examples generated by different algorithms and use them to enhance the initial pattern set.

3.4 Simulation Pattern Generation

Following the previous section, several strategies to generate expressive simulation patterns are formulated in this section. Two types of patterns are used as the basis: *random patterns* which are random values generated with equal probability of 0 or 1 for each primary input, and *stuck-at patterns* which are generated by trying to distinguish each node from constant functions 0 and 1. Generating random patterns is straightforward. The procedure to generate stuck-at patterns is described in Section 3.4.1. Then, in Section 3.4.2, an observability-based method to strengthen stuck-at patterns is elaborated. Finally, a bit-packing method to compress the pattern set is explained in Section 3.4.3.

3.4.1 Stuck-at Values

In random simulation, the possibility of a certain bit value (0 or 1) appearing in the simulation signature of some nodes in the network may be relatively low. For example, a 2-input AND gate only produces 1 when both of its fanins are 1, which is of 25% possibility if the fanin values are randomly assigned. However, a value of 1 at this node may be necessary for disproving some non-equivalence. Thus, we refine the set of simulation patterns by asserting that every node has both values appearing in its simulation signature. If only one value occurs, a new simulation pattern is created by solving a SAT problem, which forces the node to have the other value. This procedure is described in Algorithm 3.1, named *StuckAtCheck*.

In lines 1-2, we start with a small set of random simulation patterns and simulate the network to get the initial simulation signatures of each node. A SAT solver is also initialized and loaded with the CNF clauses translated from the network in lines 3-4. Then, for each node in the network (line 5), if 0 or 1 does not appear (line 6), we try to generate a pattern by assuming the missing value and solving the SAT instance (lines 7-11). If the solver finds a satisfying assignment, the desired pattern is generated (lines 12-13). In an un-optimized network, there may be nodes that never take one of the values and the solver will conclude that the problem is unsatisfiable (line 14). These nodes can be replaced by a constant node in line 15. If the solver times out or a given conflict limit is exceeded, we simply skip the node and continue

Algorithm 3.1: *StuckAtCheck:* Expressive simulation pattern generation based on stuck-at values.

```
Input: A logic network $N$
Output: A set $S$ of expressive simulation patterns
1  $S \leftarrow$ A small set of random patterns
2  $N.simulate(S)$
3  initialize $Solver$
4  $Solver.generate\_CNF(N)$
5  foreach $node\ n\ in\ N$ do
6      if $n.signature = \vec{0}$ or $n.signature = \vec{1}$ then
7          if $n.signature = \vec{0}$ then
8              $Solver.add\_assumption(n)$
9          else
10             $Solver.add\_assumption(\neg n)$
11         $result \leftarrow Solver.solve()$
12         if $result =$ SAT then
13             $S \leftarrow S \cup \{Solver.pi\_values\}$
14         else if $result =$ UNSAT then
15             Replace $n$ with constant node.
16 return $S$
```

the process with the next node.

The pattern set can be further strengthened by assuring both values appear multiple times (for example, at least 10 times) in the signature of every node. This can be done by running the SAT solver multiple times while making sure it takes different computation paths.

An example is shown in Figure 3.2. In this example, a simulation pattern is a value assignment to $\vec{x} = (a, b, c)$. Suppose there are two random patterns in the initial set $S = \{000, 110\}$. A simulation signature of a node is the bit-string of simulation results under each pattern in S, in the same order. After simulation, the simulation signature obtained for node n is 00, where 1 does not appear. Hence, by asserting $n = 1$ and solving SAT, procedure *StuckAtCheck* generates a new pattern 011 and adds it to the end of S. Now, the simulation signature of n is 001.

3.4.2 Observability

Due to the existence of observability don't cares, there may be some simulation patterns that are unobservable with respect to an internal node; these patterns are possibly less useful in disproving non-equivalence. Here, two cases are identified where a generation or re-generation of an observable pattern may be done:

- Case 1: In *StuckAtCheck* when a node is stuck at a value, and a new pattern is generated to express the other value, but this pattern is not observable.

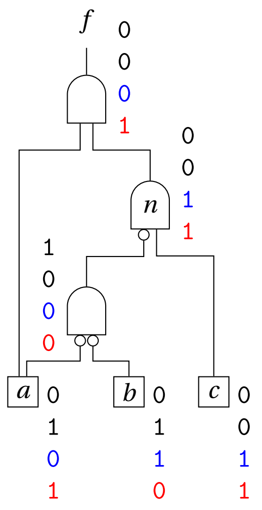


Figure 3.2: Example network for pattern generation methods.

- Case 2: A node assumes both values, but for all the patterns under which the node assumes one of the values, it is not observable.

The first case is identified during *StuckAtCheck*. Whenever a new pattern is generated (line 13), its observability with respect to the node n is checked according to the definition (Equation (2.11) in Section 2.3.2) with the following steps:

1. Simulate the network to obtain the PO values under this pattern.
2. Flip the simulation value at the output of n and simulate its TFO cone again.
3. Check if all of the PO values remain the same. If so, the pattern is unobservable.
4. Restore the value of n and simulate the TFO cone again.

This procedure is similar to how observability don't cares are computed. Step 4 is only needed if the data storage of simulation signatures is shared and reused across different procedures throughout the pattern generation process, which practically enhances efficiency by reducing re-simulations.

The second case is checked after procedure *StuckAtCheck* is completed. We iterate over all the nodes in the network again and check if, for each node, there are at least two patterns that are observable with respect to the node and the node assumes 0 and 1 respectively under the two patterns. The procedure to check whether each pattern is observable is the same as described above.

To *resolve* unobservable patterns, a procedure *ObservablePatternGeneration* is devised, which generates an observable simulation pattern $\vec{x}$ with respect to a given node n and makes sure

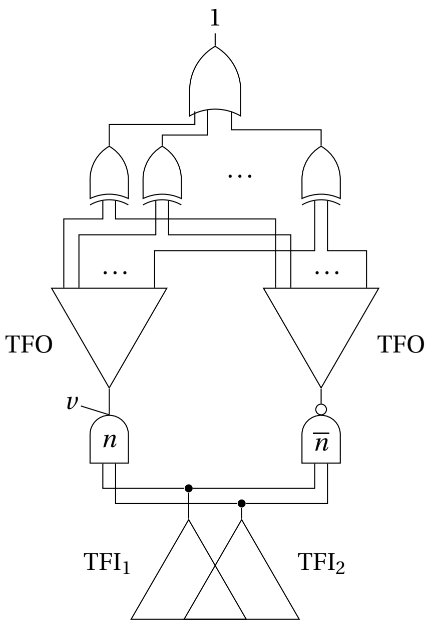


Figure 3.3: Corresponding network of the CNF instance to be built in procedure *ObservablePatternGeneration.*

that n expresses a specified value v under $\vec{x}$. This procedure builds a CNF instance, whose corresponding network is shown in Figure 3.3. In Figure 3.3, the lower two triangles TFI$_1$ and TFI$_2$ are the TFI cones of the two fanins of node n. $\overline{n}$ is created and connected to the same TFI cones as n. The TFO cone of n is duplicated (the upper two triangles) and the counterpart is connected to $\overline{n}$. Primary outputs in the two TFO cones are matched and connected to XOR gates, and the XOR gates are fed to an OR gate, forming a miter. The output value of the miter is asserted to be 1 and the output value of node n is asserted to be v. Then, the CNF instance is solved by a SAT solver. If the instance is SAT, an observable pattern is generated (Lemma 3.1), and we say that the originally unobservable pattern is *resolved.* Otherwise, if the solver returns UNSAT, n is found to be unobservable with value v and can be replaced by the constant node in the respective polarity (Lemma 3.2).

Lemma 3.1. *A satisfying input assignment $\vec{x}$ in the network of Figure 3.3 is an observable pattern with respect to node n.*

Proof. By definition, $\vec{x}$ is observable with respect to n if the value of at least one of the primary outputs of the network under $\vec{x}$ is different when n is replaced by $\overline{n}$. This condition is ensured by the miter of the TFO cones of n and $\overline{n}$ in Figure 3.3. □

Lemma 3.2. *If a node n is never observable with value v ($v \in \{0,1\}$), then it can be replaced by constant $\neg v$ without changing the network function(s). That is, there does not exist a primary input assignment $\vec{x}$, such that one of the primary outputs has different values in the original network and in the modified network.*

Proof. Assume the opposite: there exists a primary input assignment $\vec{x}$, such that at least one of the primary outputs has a different value after replacing n with $\neg v$. If the value of n is $\neg v$ under $\vec{x}$, all node values in the network, including primary outputs, remain unchanged if n is replaced by $\neg v$. If the value of n is v under $\vec{x}$, because n is not observable with v, all primary outputs remain at the same value when the node value of n changes to $\overline{n} = \neg v$, which contradicts the assumption. □

In order to limit the computation in large networks, the TFO in Figure 3.3 is practically restricted to a depth. In this case, all the leaves of the cone should be XOR-ed with their counterparts to build the miter. Note that restricting the TFO depth weakens the definition of observability, but is essential for scalability. Empirically, using a depth of 5 is shown to be a good tradeoff between quality and runtime.

After an observable pattern $\vec{x}$ is generated, in Case 1, we can replace the pattern generated by *StuckAtCheck* with $\vec{x}$. In Case 2, we simply add $\vec{x}$ to the set of patterns.

We continue with the example in Figure 3.2 with three patterns in the set $S = \{000, 110, 011\}$. By checking the observability of each pattern, it is found that only 110 is observable and the value of n under this pattern is 0. Hence, procedure *ObservablePatternGeneration* generates another pattern 101 making $n = 1$. This pattern is indeed observable because flipping the value of n from 1 to 0 also makes the PO value f change from 1 to 0.

3.4.3 Bit-Packing

For some large benchmarks with many primary inputs, the size of the generated pattern set can be large, slowing down simulation. In the field of ATPG, test patterns are often compressed by first identifying *care* and *don't-care* bits in them [MK06]. The set of care bits in a test pattern is the set of PI values that contribute to detecting a certain fault, while the don't-care bits are the PIs that can be assigned to any value. We integrated a similar technique in our simulation pattern generation.

Similar to test pattern compression, the care bits in a simulation pattern are the PI values that contribute to proving that the node is not stuck-at and in fact observable at one of the outputs. During simulation pattern generation with the previously described methods, care bits are identified by a simple structural support analysis, which highlights control paths from the inputs to the target node, and from the target node to at least one output where it is observed.

After generating several patterns, the pattern set is compressed by trying to pack each new pattern into one of the preceding patterns. Two patterns can be packed together if their care bits do not overlap. To pack a pattern p_1 into another pattern p_2, the care bits of p_1 are written into don't-care bits of p_2, and these bits are marked as cares in p_2.

3.4.4 Discussion

In this section, we illustrate methods to derive an initial set of expressive patterns serving as the basis of the simulation-guided paradigm. Starting from a mixture of random patterns and stuck-at patterns as the basis and depending on the computation effort taken by the pattern generation phase, observability checks can be applied to strengthen or append the pattern set. It may seem, from the algorithms, that each pattern is generated for a specific node in the network, which may be removed later during logic optimization and the pattern becomes useless. However, we argue that this is not a problem because even random patterns play an important role in this paradigm, as shown in our experimental results. Moreover, it is practically inefficient to keep track of which pattern is generated for which node and which patterns are still useful, especially after bit-packing. As another piece of evidence, our experimental results on ECO benchmarks show that the generated patterns are as useful for a functionally modified network even if they are generated with the original version of the design.

3.5 Simulation-Guided Resubstitution

In this section, the simulation-guided paradigm is demonstrated with Boolean resubstitution as an example application in logic synthesis. The main difference of our algorithm, compared to a state-of-the-art resubstitution algorithm [MB06], is in the representation of the divisors. Instead of using the complete truth table of the *local* function of the node, we use the simulation signature approximating the *global* function of the node. The algorithm consists of the following steps:

1. Generation of a set of expressive simulation patterns, as described in Section 3.4.
2. Simulation of the network with these patterns to obtain simulation signatures for each node.
3. Iterating over all nodes in the network and calling the currently chosen node the root node. Estimating the gain by computing the root node's MFFC and collecting the divisors. Skipping the node if the gain is too small or if there are no divisors. Details of this step are described in Section 2.3.1.
4. Searching for resubstitution candidates in terms of dependency functions using simulation signatures. Details of this step are described in Chapter 4.
5. Validating the resubstitution with SAT solving by assuming non-equivalence. An UNSAT result validates the resubstitution, while a SAT result provides an input assignment under which the optimized network is not equivalent to the original network. In the latter case, the counter-example is added to the set of simulation patterns.
6. Iterating starting from Step 3, until all nodes in the network have been processed.

Algorithm 3.2: *SimResub:* One iteration of Steps 4 and 5 in simulation-guided Boolean resubstitution.

Input: A root node n in a simulated network N, its MFFC *MFFC*, and a set G of divisors
Output: A legal (verified) candidate to substitute n, if exists

```
1  initialize Solver
2  Solver.generate_CNF(N)
3  while TRUE do
4      H ← resynthesize(n, G, min{|MFFC|, m})
5      if H ≠ NULL then
6          result ← Solver.verify(n, H)              // Detailed in Algorithm 3.3
7          if result = TRUE then
8              return H
9          else if result = FALSE then
10             N.re_simulate()
11         else
12             break
13     else
14         break
15 return NULL
```

Simulation of the entire network in Step 2 enables better incorporation of global satisfiability don't cares without extra cost, which allows more optimization potential compared to the windowing-based approach as in [MB06]. The collection of counter-examples in Step 5 expands the simulation pattern set, which further improves the efficiency of later optimization runs. In the remainder of this section, we focus on Steps 4 and 5, shown in Algorithm 3.2, which differ the most.

A SAT solver is initialized and the CNF clauses encoding gate logic are generated and added to the solver in lines 1-2. In line 4, a simulation-signature-based resynthesis algorithm is used to find a dependency circuit of up to m^* nodes, where m^* is the smaller value among a user-specified parameter m and the size of the MFFC. Procedure *resynthesize* heuristically searches for a minimum-node AIG implementation H of the target function f_t using a set of divisors G as PIs. Both the target function and the divisors are represented by their simulation signatures. The PO of H should have the same signature as the given target f_t. Details of the underlying algorithm are described in Chapter 4.

Since the simulation signatures are an approximation of the node's function, the resubstitution candidate needs to be formally verified. Procedure *verify* in line 6 uses the SAT solver to try to find a pattern, under which nodes n and H_{out} have different values. This is detailed in Algorithm 3.3. The resubstitution is legal if the solver returns UNSAT (lines 4-5 in Algorithm 3.3 and lines 7-8 in Algorithm 3.2); otherwise, a new pattern is added to the set and the network is re-simulated if the solver returns SAT (lines 6-8 in Algorithm 3.3 and lines 9-10 in Algorithm 3.2). Note that if the simulation signatures are stored as sequences of multiple machine words, a new

Algorithm 3.3: *Solver.verify:* Verify a resubstitution candidate using a SAT solver.

Input: A root node n in a simulated network N, and a dependency circuit H with some nodes in N as PIs and H_{out} as PO

Output: Whether it is legal to substitute n with H

```
1 Solver.generate_CNF(H)
2 Solver.add_assumption( literal(n) ⊕ literal(H_out) )
3 result ← Solver.solve()
4 if result = UNSAT then
5     return TRUE
6 else if result = SAT then
7     N.add_pattern(Solver.pi_values)
8     return FALSE
9 return UNKNOWN
```

pattern is appended to the end of the last word and only this word needs to be re-computed because the other words remain the same. With the appended signatures, *resynthesize* gives a different result in the next invocation. The process continues until one resubstitution is validated (lines 7-8), or the SAT solver times out (lines 11-12), or until the engine cannot find another candidate dependency function (lines 13-14).

3.6 Simulation-Guided Equivalence Checking

CEC after logic synthesis can benefit from the simulation information collected and used for logic optimization. This is because, in the process of CEC [Mis+06a], one of the major tasks is disproving candidate equivalences, which relies on SAT-solving when counter-examples cannot be easily found with random simulation. The pre-computed expressive simulation patterns provided to the CEC engine can be used to disprove many of the non-equivalent nodes directly without any SAT-solving.

The command `&cec` in ABC[1] [BM10], which is an improved version of `cec` [Mis+06a], compares AIGs derived from two versions of the design presented for CEC. Internally, it generates random simulation patterns iteratively to detect candidate equivalent pairs and filter out non-equivalent nodes. Random simulation is repeated until no more refinement can be made, i.e., no more non-equivalent nodes being distinguished. Then, a SAT solver is called to formally prove the equivalence pairs by assuming non-equivalence, similar to the verification procedure in the resubstitution algorithm presented in the previous section. If the solver returns UNSAT, the equivalence pair is formally proved; otherwise, if the solver returns SAT, a counter-example is generated. The counter-example disproves the given candidate equivalence and potentially other unproven ones.

We implemented simulation-guided CEC by modifying command `&cec` to use pre-generated

[1]Available: github.com/berkeley-abc/abc

patterns instead of generating random patterns. This can be useful when the design is optimized with the proposed paradigm, for example, the simulation-guided resubstitution developed in this chapter, so that an expressive set of patterns pre-generated, and maybe even supplemented with the counter-examples generated during optimization, is already in hand. Without any extra cost, the patterns can be reused in CEC to reduce SAT calls disproving equivalence.

3.7 Experimental Results

In Sections 3.7.1 and 3.7.2, we first investigate the expressiveness of simulation patterns generated using different methods by comparing the number of counter-examples encountered in resubstitution. After finding a good strategy, we use it to generate a pattern set to be used for other experiments and report its size before and after bit-packing in Section 3.7.3. Then, Section 3.7.4 demonstrates how an expressive pattern set makes a shift in runtime from optimization to pattern generation, and Section 3.7.5 confirms the reusability of patterns for functionally-modified networks with a set of ECO benchmarks. Finally, the advantages of simulation-guided resubstitution and simulation-guided equivalence checking are shown in Sections 3.7.6 and 3.7.7, respectively.

The experiments are performed on a Linux machine with Xeon 2.5 GHz CPU and 256 GB RAM. The OpenCore designs from IWLS'05 benchmark[2] are used in all experiments, except for those in Section 3.7.5. When generating the patterns and testing the quality of resubstitution and equivalence checking in Sections 3.7.3, 3.7.4, 3.7.6 and 3.7.7, the benchmarks are preprocessed with redundancy removal by iterating command `ifraig` in ABC until no reduction in size. The results for the preprocessed benchmarks are reported in Table 3.1. The preprocessed benchmarks and the simulation patterns used can be found online[3].

3.7.1 Size of Simulation Pattern Set

Intuitively, the more simulation patterns used, the higher the chance that the paradigm saves time by not attempting to prove non-equivalences, i.e., a larger set of simulation patterns is expected to be more expressive. Following the definition of expressive patterns in Section 3.3, we measure the *expressive power* of a pattern set using the percentage decrease, as compared to a baseline set, in the number of counter-examples encountered in resubstitution, which is calculated separately for each benchmark. Different from the resubstitution framework described in Section 3.5, the counter-examples are not added to the simulation set, to isolate the impact of the provided patterns.

We start by investigating the expressive power of random patterns based on their count. In Figure 3.4, each bar represents how expressive a pattern set of the respective size is, compared

[2]Available: iwls.org/iwls2005/benchmarks.html
[3]Available: github.com/lsils/sim-LSV_exp

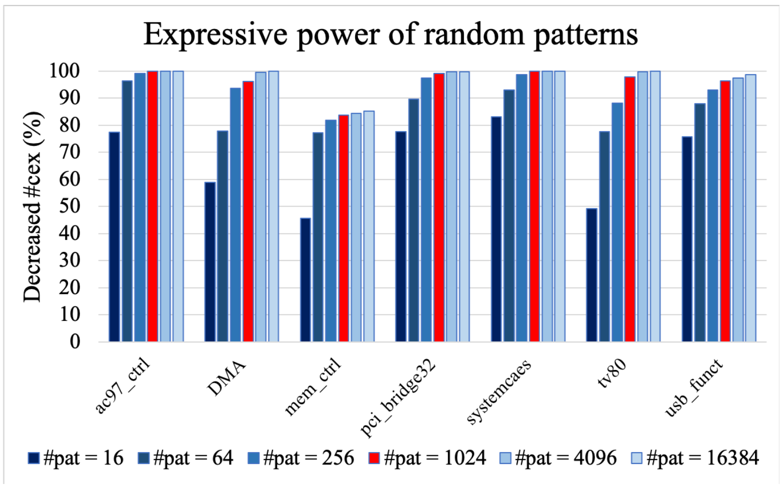


Figure 3.4: Decreased percentages of counter-examples when provided with different number ($\#pat$) of random simulation patterns, compared to the baseline $\#pat = 4$.

to the baseline of using only four simulation patterns. The smaller sets are subsets of the larger sets to avoid the biasing effect of randomness. Since the trend is similar for each benchmark, only some medium-sized benchmarks (with around 10 to 20 thousand nodes) are shown here. As the size grows by a factor of four (leading to 4, 16, 64, etc. patterns), the expressive power increases very fast at first, as expected, but saturates at a few hundred to a few thousand patterns. Fortunately, a thousand patterns is still a practical size, for which bit-parallel simulation runs fast.

A similar phenomenon is observed when patterns are generated by *StuckAtCheck*. As discussed in Section 3.4.1, additional patterns can be used to ensure that every node has at least b bits of 0 and b bits of 1 in its signature. In the following experiments, stuck-at patterns are abbreviated as "`s-a`", with a prefix "bx" listing parameter b. In Figure 3.5, since the stuck-at pattern counts are different for each benchmark, the pattern set size is normalized to the network size and plotted in the logarithmic scale. Only benchmarks that are smaller than 25k nodes are included. The baseline pattern set is "`1x s-a`". It is observed that larger sets of patterns are usually more expressive. Note that randomness plays a role in this case, since the default variable polarities, which determine initial variable values in the SAT solver, are randomly reset before each run.

3.7.2 Pattern Generation Strategies

In this section, the expressive power of simulation patterns generated by *StuckAtCheck* is compared with the case when observability is used (suffix "`-obs`") and/or when an initial random pattern set of size 256 is used (prefix "`rand 256`").

The observability check and observable pattern generation are done with a fanout depth of 5 levels. A conflict limit of 1000 is set for the SAT solver, and there is no time-out limit set. A set of 256 random patterns is used as the baseline in Figure 3.6. Four small benchmarks, for

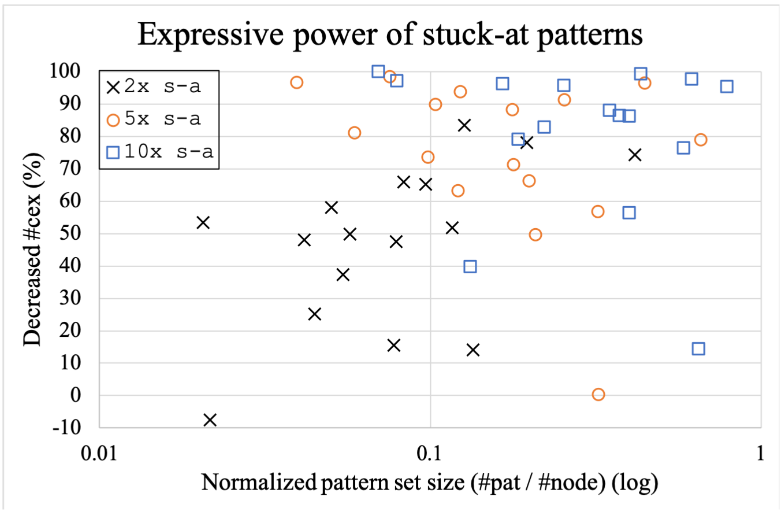


Figure 3.5: Decreased percentages of counter-examples when using different sets of stuck-at simulation patterns, compared to the baseline set "`1x s-a`".

which the random pattern sets are more expressive than "`1x s-a`" and/or "`1x s-a-obs`", are not shown in the figure. Larger benchmarks with more than 25k nodes are also excluded. The geometric means of the sizes of the pattern sets are 143 for "`1x s-a`", 244 for "`1x s-a-obs`", 354 for "`rand 256 + 1x s-a`" and 462 for "`rand 256 + 1x s-a-obs`". On the other hand, the geometric means of the decreased percentages of the counter-examples are 91.3%, 96.5%, 97.1%, and 99.5%, respectively.

It is observed that patterns generated by *StuckAtCheck* are usually more expressive than random patterns, except for a few, typically small, benchmarks. Also, using observability increases the expressive power of the generated patterns. Finally, seeding the pattern generation engine with an initial set of random patterns not only speeds up the generation process but also makes the resulting patterns more expressive.

As the patterns generated with "`rand 256 + 1x s-a-obs`" are shown to be the most expressive, these pattern sets are used in the following experiments in Sections 3.7.3, 3.7.4, 3.7.6 and 3.7.7. Table 3.1 lists some information on the benchmarks and their pattern sets. On average, about 80% of the runtime (about 50% for the largest five benchmarks) in pattern generation was spent in the observability-based methods, including time for checking if a pattern is observable, SAT-solving with the TFO cone, and re-simulation after a new pattern is generated. As seen in Figure 3.6, using observability increases the expressive power of the generated patterns, but not much. Thus, in practice, one may consider disabling observability awareness for larger benchmarks. There is no constant node detected because the benchmarks are preprocessed with redundancy removal, and there are about 0.1% unobservable nodes found, on average.

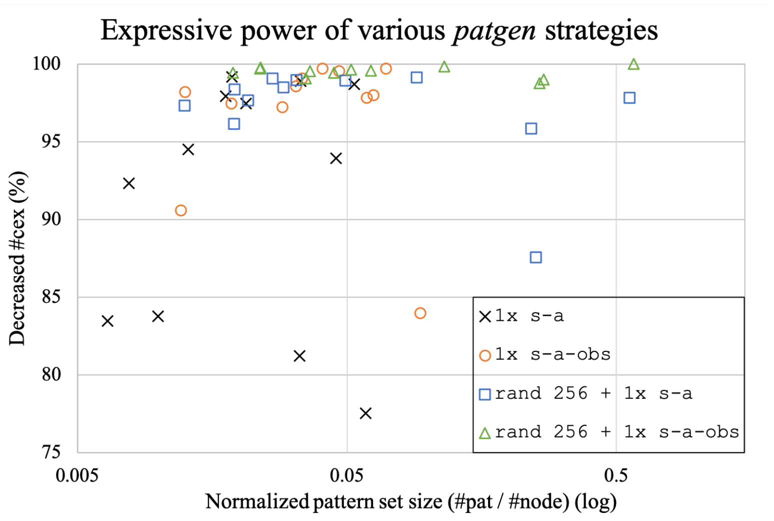


Figure 3.6: Decreased percentages of counter-examples when using pattern sets generated with different strategies, compared to the baseline set "`rand 256`".

3.7.3 Pattern Compression with Bit-packing

As discussed in Section 3.4.3, the generated patterns can be packed together to reduce the pattern set size and speed up the simulation. This technique becomes more important in larger benchmarks with huge amounts of primary inputs. The middle part of Table 3.1 shows the total number of generated patterns (column *gen.*), the final number of patterns after bit-packing (column *packed*), and the ratio of the two sizes (column *(%)*). The 256 random patterns are not bit-packed, nor included in this table. On average, the sizes of the packed pattern sets are about 70% of the original sets.

3.7.4 Effect of Expressive Patterns in Resubstitution

As stated in the motivation, an expressive set of simulation patterns is used to shift the computation effort from the optimization algorithms to pattern pre-computation. Table 3.2 shows how the quality of the patterns affects the runtime of pattern generation (*patgen*) and resubstitution (*resub*). For simplicity, only some of the larger benchmarks with more obvious effects are shown in this table. A better set of patterns (Table 3.2, "`rand 256 + 1x s-a-obs`") efficiently filters out many illegal resubstitutions without calling the SAT solver, resulting in the reduced counter-example counts (#cex) and faster runtimes. Note that there is no difference in optimization quality (i.e., circuit size reduction) caused by using different patterns because if an illegal resubstitution is not filtered out by simulation signatures, it is still disproved by SAT solving.

Furthermore, in practice, when the same design is repeatedly synthesized during development or when simulation patterns are reused by different optimization engines, counter-examples from the previous runs can be saved for later use. In this case, the additional counter-example

Table 3.1: Number of generated patterns before and after bit-packing.

benchmark			#patterns		ratio	runtime
name	size	#PIs	gen.	packed	(%)	(s)
leon2	787972	298888	23526	14858	63.2	17080.75
leon3_opt	972952	370159	24820	16448	66.3	24566.67
leon3	1085718	370159	24739	16161	65.3	23471.45
leon3mp	650722	217858	13799	9483	68.7	5045.94
netcard	802846	195730	28206	13944	49.4	8896.10
ac97_ctrl	14199	4482	88	27	30.7	0.38
aes_core	21441	1319	163	18	11.0	0.74
des_area	4827	496	18	18	100.0	0.19
des_perf	81998	17850	54	54	100.0	3.95
DMA	21992	5070	886	384	43.3	2.11
DSP	44132	7835	1374	736	53.6	6.87
ethernet	86293	21216	2787	1340	48.1	27.59
i2c	1120	275	65	57	87.7	0.02
mem_ctrl	7870	2281	601	393	65.4	0.70
pci_bridge32	22521	6880	714	207	29.0	1.82
RISC	73789	15678	3139	1012	32.2	17.30
sasc	770	250	1	1	100.0	0.00
simple_spi	1034	280	32	25	78.1	0.01
spi	3762	505	184	184	100.0	0.18
ss_pcm	405	193	2	2	100.0	0.00
systemcaes	12108	1600	39	38	97.4	0.23
systemcdes	2857	512	3	3	100.0	0.07
tv80	9091	732	408	404	99.0	0.55
usb_funct	15245	3620	643	238	37.0	0.92
usb_phy	440	211	9	8	88.9	0.00
vga_lcd	126427	34247	5142	2957	57.5	120.34
wb_conmax	47449	2670	206	170	82.5	1.60

count during later runs can go down to nearly zero, and the runtime is only spent on logic synthesis or verification tasks, such as proving equivalences among the nodes or computing dependency functions and validating them. The latter scheme will be verified in the next section and be used from then on.

3.7.5 Reusability of Simulation Patterns

In support of our assumption, the reusability of the generated patterns and the counter-examples are verified with a set of ECO benchmarks [KJR20]. For each design, there is an old version and a new version which are functionally different. The results of two runs of resubstitution with the two versions of benchmarks are reported and compared in Table 3.3. First, a set of patterns is generated for the old version with "`rand 256 + 1x s-a-obs`" where only the first case of observability check is performed. Columns A and B show the number of counter-examples (#cex) and the runtime of resubstitution on the two versions of benchmarks using this generated pattern set. Comparing them, it is observed that the patterns are as effective on the new benchmarks, even though they are generated with the old ones. In columns C

Table 3.2: Resubstitution runtime as a function of the number of counter-examples produced.

	`rand 256`			`rand 256 + 1x s-a-obs`		
		runtime (s)			runtime (s)	
benchmark	#cex	*patgen*	*resub*	#cex	*patgen*	*resub*
aes_core	69	0.01	0.72	7	0.74	0.34
des_perf	11	0.01	3.23	2	3.95	3.50
DMA	4923	0.01	2.15	440	2.11	0.41
DSP	8436	0.01	5.71	510	6.87	1.71
ethernet	50334	0.01	67.27	5329	27.59	10.63
pci_bridge32	3303	0.01	2.61	484	1.82	0.96
RISC	15052	0.01	16.02	589	17.30	2.81
vga_lcd	88008	0.01	182.36	3749	120.34	13.16
wb_conmax	920	0.01	0.66	146	1.60	0.64

and D, resubstitution is performed again, but using the generated patterns appended with the counter-examples collected in A. There are almost no new counter-examples in column C when the same optimization algorithm is applied on exactly the same benchmarks, as expected. Moreover, when applying on slightly different networks in column D, the number of counter-examples is reduced by 73% compared to the first run (B). The runtime in D is only slightly higher than in C, showing that most of the runtime is spent on computing dependency functions and validating the legal resubstitutions, which are inevitable. The lower-right column compares a flow optimizing first the old networks and then the new ones without learning of counter-examples (A+B) against one that learns the counter-examples from previous runs (A+D).

3.7.6 Quality of Simulation-Guided Resubstitution

This section shows the improvements in terms of resubstitution quality. Tables 3.4 and 3.5 compare the proposed framework with command `resub` [MB06] in ABC [BM10], which performs truth-table-based resubstitution. Because computing simulation patterns in our framework results in detecting combinational equivalences [Mis+05], for a fair comparison, the benchmarks are preprocessed by repeating the command `ifraig` in ABC until no more size reduction is observed. The quality of results, presented in the *gain* columns, is measured with the reduction percentage in network size after optimization, i.e., the difference in the number of nodes before and after resubstitution, divided by the original network size. Simulation patterns used in our framework are initially generated with "`rand 256 + 1x s-a-obs`", bit-packed (as described in Section 3.4.3), and then incrementally supplemented with the counter-examples generated from the previous runs of the same resubstitution settings in each column. After the resubstitution run in the last column of Table 3.5, the sizes of pattern sets increase by 30% on average.

Two parameters can be set in both flows: the maximum cut size k used to collect divisors in

Table 3.3: Resubstitution efficiency after ECO with or without counter-example learning.

benchmark version		A old		B new		C old		D new	
pattern set		generated		generated (for old)		with CEX from A		with CEX from A	
benchmark	size	#cex	time (s)	#cex	time (s)	#cex	time (s)	#cex	time (s)
design1	218679	2711	30.92	2869	31.75	0	12.39	441	17.63
design2	344	16	0.01	11	< 0.01	0	0.00	11	< 0.01
design3	453920	3089	48.52	3006	46.76	0	19.56	219	23.37
design4	30819	579	0.81	594	0.82	0	0.36	150	0.55
design5	3582	76	0.04	63	0.04	0	0.03	6	0.03
design6	77555	1161	4.40	1180	4.51	0	2.24	126	2.78
design7	62336	844	2.10	907	2.18	1	1.13	123	1.41
design8	20517	540	0.59	575	0.65	0	0.31	130	0.46
design9	4650	69	0.05	83	0.05	0	0.03	26	0.04
design10	15995	86	0.23	138	0.21	0	0.18	71	0.20
design11	48817	949	2.17	931	2.10	0	1.06	94	1.18
average		920.00	8.17	941.55	8.10	0.09	3.39	127.00	4.33

	B vs. D		(A+B) vs. (A+D)	
benchmark	Δ#cex	Δtime	Δ#cex	Δtime
design1	-84.63%	-44.47%	-43.51%	-22.53%
design2	0.00%	N/A*	0.00%	0.00%
design3	-92.71%	-50.02%	-45.73%	-24.55%
design4	-74.75%	-32.93%	-37.85%	-16.56%
design5	-90.48%	-25.00%	-41.01%	-12.50%
design6	-89.32%	-38.36%	-45.02%	-19.42%
design7	-86.44%	-35.32%	-44.77%	-17.99%
design8	-77.39%	-29.23%	-39.91%	-15.32%
design9	-68.67%	-20.00%	-37.50%	-10.00%
design10	-48.55%	-4.76%	-29.91%	-2.27%
design11	-89.90%	-43.81%	-44.52%	-21.55%
average	-72.99	-32.39*	-37.25	-14.79

*The runtime is too fast to compute the reduction rate, hence this benchmark is excluded from the average.

Table 3.4: Resubstitution quality on AIGs comparing against ABC's `resub` command. Baseline: at most one node insertion.

		ABC $k = 10, m = 1$		Ours, $k = 10, m = 1$		Ours, $k = 100, m = 1$	
benchmark	size	gain (%)	time (s)	gain (%)	time (s)	gain (%)	time (s)
leon2	787972	0.11	69.48	0.13	65.52	0.32	1639.16
leon3_opt	972952	0.18	55.40	0.23	82.55	0.28	1113.55
leon3	1085718	0.10	55.11	0.11	90.25	0.19	1347.85
leon3mp	650722	0.08	30.16	0.10	41.04	0.19	406.59
netcard	802846	0.08	52.79	0.09	60.21	0.13	1062.90
ac97_ctrl	14199	1.25	0.15	1.25	0.08	1.27	0.10
aes_core	21441	1.50	0.42	1.60	0.48	2.32	2.59
des_area	4827	1.82	0.08	2.15	0.07	2.15	0.50
des_perf	81998	6.07	1.37	7.01	2.91	7.17	3.61
DMA	21992	0.89	0.27	1.04	0.20	1.29	1.12
DSP	44132	2.13	0.54	2.71	0.64	3.32	4.08
ethernet	86293	0.31	2.03	0.34	1.95	0.49	15.76
i2c	1120	4.29	0.01	5.09	0.01	7.68	0.02
mem_ctrl	7870	1.91	0.08	3.44	0.07	5.17	0.89
pci_bridge32	22521	0.78	0.40	0.86	0.26	1.19	0.76
RISC	73789	1.83	0.71	2.18	0.91	4.21	3.94
sasc	770	0.65	< 0.01	0.65	< 0.01	0.65	< 0.01
simple_spi	1034	1.74	0.01	1.64	0.01	2.22	0.01
spi	3762	2.15	0.07	2.23	0.04	2.37	0.36
ss_pcm	405	0.25	< 0.01	0.25	< 0.01	0.25	< 0.01
systemcaes	12108	0.30	0.11	0.40	0.10	0.45	0.48
systemcdes	2857	4.83	0.04	5.50	0.06	5.67	0.23
tv80	9091	2.41	0.15	2.85	0.13	4.93	2.67
usb_funct	15245	2.93	0.16	3.67	0.14	7.65	0.35
usb_phy	440	2.73	< 0.01	3.64	< 0.01	3.64	< 0.01
vga_lcd	126427	0.09	5.07	0.12	4.59	0.14	51.31
wb_conmax	47449	1.19	0.78	9.59	0.67	9.59	1.99
average		1.58	10.20	2.18	13.07	2.78	209.66
geomean		0.81	0.38*	1.02	0.39*	1.35	1.81*

*The values smaller than 0.01 are replaced with 0.005 when calculating geomean.

the TFI of the root node and the maximum number m of nodes in the dependency circuit.[4] Since [MB06] relies on computing truth tables in the window, $k \leq 10$ is typically used as a reasonable trade-off between efficiency and quality. In contrast, windowing in our framework is applied only to avoid potential runtime blow-up for large benchmarks and k can be set to arbitrarily large values when a longer runtime is acceptable.

When the algorithms are limited to at most one node insertion ($m = 1$), Table 3.4 shows that our framework achieves 2.18% network size reduction on average using the same, small window size ($m = 10$), comparing to 1.58% by the state-of-the-art. This improvement is due to better consideration of global satisfiability don't-cares. Moreover, we can arbitrarily extend

[4]In ABC's command line interface, cut size is set by argument `-K` and the maximum number of nodes in the dependency circuit is set by argument `-N`. However, here, we use the same symbols as in the other parts of the thesis for consistency.

Table 3.5: Resubstitution quality on AIGs comparing against ABC's `resub` command. Best achievable quality.

		ABC $k = 16, m = 3$		Ours, $k = 100, m = 20$	
benchmark	size	gain (%)	time (s)	gain (%)	time (s)
leon2	787972	0.35	1811.35	0.65	5984.96
leon3_opt	972952	0.73	1273.16	1.02	5462.90
leon3	1085718	0.28	1824.90	0.63	5239.15
leon3mp	650722	0.80	875.65	0.57	1342.39
netcard	802846	0.28	1562.19	0.56	5425.19
ac97_ctrl	14199	2.24	4.81	6.87	0.93
aes_core	21441	3.02	19.53	6.29	8.62
des_area	4827	3.09	3.50	5.72	1.08
des_perf	81998	8.70	74.10	15.78	7.32
DMA	21992	1.93	8.49	2.78	3.36
DSP	44132	4.14	48.02	5.74	13.92
ethernet	86293	0.95	106.04	2.72	74.15
i2c	1120	8.48	0.56	11.88	0.13
mem_ctrl	7870	4.08	3.67	8.93	2.64
pci_bridge32	22521	2.33	17.52	2.78	3.27
RISC	73789	3.47	56.22	7.56	17.04
sasc	770	1.56	0.13	1.82	0.02
simple_spi	1034	4.64	0.35	5.32	0.06
spi	3762	3.19	2.16	5.24	0.74
ss_pcm	405	0.99	0.03	1.23	< 0.01
systemcaes	12108	0.64	11.04	1.68	2.16
systemcdes	2857	7.46	1.87	11.41	0.28
tv80	9091	5.26	8.62	11.75	7.34
usb_funct	15245	7.04	7.56	11.82	1.96
usb_phy	440	7.73	0.07	10.91	0.01
vga_lcd	126427	0.26	207.27	0.48	153.19
wb_conmax	47449	14.95	48.41	17.15	6.54
average		3.65	295.45	5.90	879.98
geomean		2.13	14.72	3.55	6.15*

*The value smaller than 0.01 is replaced with 0.005 when calculating geomean.

the window size and achieve up to 2.78% gain when a longer runtime is acceptable.

In Table 3.5, parameters in `resub` are set to their extreme values ($k = 16, m = 3$), and parameters in our framework are set to large values semantically close to infinity. It is observed that our framework can achieve up to 5.90% reduction while 3.65% is the best `resub` can do, and the improvement comes even with faster runtime in most of the benchmarks. The reason why our framework is especially slow in the largest five benchmarks is because they also have large numbers of primary inputs and large sizes of pattern sets (shown in Table 3.1), which slow down simulation as well as the computation of dependency functions. This can be ameliorated, however, by fine-tuning the trade-off between quality and runtime according to the user's needs.

Furthermore, the proposed framework is also shown to be applicable on 2-LUT networks, or

Table 3.6: Resubstitution quality on XAGs comparing against ABC's `&mfs` command.

		`abc> &mfs -a`		Ours, $k = 10, m = 1$	
benchmark	size	gain (%)	time (s)	gain (%)	time (s)
leon2	785623	0.12	612.10	0.11	103.89
leon3_opt	970570	0.13	697.90	0.21	139.80
leon3	1082547	0.10	705.60	0.09	139.79
leon3mp	649333	0.13	317.60	0.09	59.96
netcard	800880	0.07	676.90	0.09	91.23
ac97_ctrl	13945	0.47	0.50	1.23	0.09
aes_core	18951	0.82	5.54	1.89	0.48
des_area	4673	1.16	2.20	2.23	0.08
des_perf	76458	3.23	11.96	7.53	2.87
DMA	21435	0.55	3.37	1.03	0.25
DSP	41795	1.06	15.97	1.90	0.55
ethernet	85355	0.17	19.00	0.30	2.06
i2c	1101	3.72	0.09	5.09	0.01
mem_ctrl	7408	4.94	1.96	3.62	0.07
pci_bridge32	21759	0.38	1.79	0.86	0.25
RISC	69514	1.72	12.79	1.46	0.90
sasc	733	0.82	0.02	0.68	0.01
simple_spi	1003	1.60	0.05	1.69	0.01
spi	3697	0.70	1.29	1.87	0.06
ss_pcm	398	0.00	0.01	0.25	0.01
systemcaes	10652	0.70	1.55	0.58	0.09
systemcdes	2744	3.72	0.62	5.69	0.07
tv80	8751	2.79	9.26	2.43	0.13
usb_funct	14201	1.88	1.00	3.15	0.13
usb_phy	408	3.19	0.01	3.43	0.01
vga_lcd	126093	0.06	56.83	0.11	5.25
wb_conmax	47449	14.38	8.75	9.59	0.63
average		1.80	117.21	2.12	20.32
geomean		N/A	3.93	0.97	0.46

essentially, XAGs. Table 3.6 compares the proposed framework with command `&mfs` [Mis+11b] in ABC.[5] The `ifraig`-preprocessed benchmarks are mapped into 2-LUT networks by the command `&if -K 2` in ABC and read in as XAGs in *mockturtle*. The simulation pattern set generated in Section 3.7.3 with the AIG benchmarks and used in the experiments in Tables 3.4 and 3.5 is reused for the XAG experiment. In Table 3.6, the numbers of 2-LUTs (or XAG nodes) are reported in column *size*, and the percentage reduction and runtime of the two algorithms are reported in columns *gain* and *time*, respectively. Using only an unaggressive parameter setting ($k = 10, m = 1$), our framework outperforms command `&mfs` in both optimization quality and efficiency.

[5]While the paper was published in 2011, the technical implementation has been continuously improved over time and there are several versions of the same concept in ABC, such as commands `mfs` and `mfs2`. Among them, `&mfs` is believed to be the newest and the best version.

Table 3.7: Efficiency of CEC with or without using expressive patterns.

		`abc> &cec`			`&cec` with expressive patterns			
benchmark	size	#SAT	#UNSAT	time (s)	#pats	#SAT	#UNSAT	time (s)
leon2	787972	8579	19738	32.32	3200	7150	19465	41.53
leon3_opt	972952	19529	50162	42.15	3200	14751	50020	49.10
leon3	1085718	113427	127162	88.64	3200	80242	127163	82.64
leon3mp	650722	65439	90482	43.78	3200	37522	84326	35.52
netcard	802846	21691	107513	31.14	3200	19269	107523	28.93
ac97_ctrl	14199	0	2215	0.19	384	41	2215	0.17
aes_core	21441	0	3177	0.71	320	2	3177	0.65
des_area	4827	0	393	0.08	320	0	393	0.07
des_perf	81998	0	5423	1.22	320	0	5423	0.99
DMA	21992	337	2981	0.45	832	298	2981	0.34
DSP	44132	911	6232	1.60	1600	249	6230	1.23
ethernet	86293	596	10505	1.19	1408	9817	10486	2.25
i2c	1120	65	165	0.03	320	33	163	0.03
mem_ctrl	7870	651	927	0.24	832	166	929	0.18
pci_bridge32	22521	612	3132	4.44	576	511	3132	4.40
RISC	73789	3638	9084	2.37	1472	500	9083	1.37
sasc	770	0	116	0.03	320	0	116	0.02
simple_spi	1034	14	157	0.03	320	24	157	0.03
spi	3762	109	469	0.12	448	160	469	0.12
ss_pcm	405	0	62	0.02	320	0	62	0.02
systemcaes	12108	0	1384	0.24	384	6	1384	0.23
systemcdes	2857	0	329	0.06	320	1	329	0.05
tv80	9091	279	1160	0.33	704	225	1160	0.27
usb_funct	15245	809	2003	0.37	512	275	2003	0.25
usb_phy	440	0	57	0.02	320	0	57	0.02
vga_lcd	126427	13852	13682	4.28	3584	1055	13670	2.28
wb_conmax	47449	2	3793	0.61	448	3	3793	0.51
average		994.32	3065.73	0.85	730.18	607.55	3064.18	0.70

3.7.7 Reduction on SAT Calls in CEC with Expressive Patterns

Finally, to show the effectiveness of the proposed paradigm on other logic synthesis and verification algorithms, we take CEC as another example. The `&cec` command in ABC [Mis+06a] is considered the state of the art. It iteratively generates random patterns for simulation to find equivalent pair candidates. This command is modified to take pre-generated patterns and use them for simulation. The number of SAT results (disproving equivalence; #SAT) and UNSAT results (proving equivalence; #UNSAT) in `&cec` with and without using pre-generated expressive patterns are reported in Table 3.7. For simulation efficiency, an upper limit of 3200 on the number of patterns is set. It can be observed from the table that the average number of SAT results is reduced by about 40%; when combined with the UNSAT results, which are unchanged, the total number of SAT solver calls is reduced by about 9.5%. In most cases, the runtime does not decrease because it is dominated by the UNSAT calls, and too many patterns slow down the simulation. Nevertheless, the runtime overhead in simulation can be mitigated if the patterns can be better compacted, or if the simulation can be speeded up (e.g., by using *Haswell New Instructions* (AVX2) which provides single-cycle bitwise operations on longer machine words) in a future implementation of simulation-guided CEC. More importantly, by showing a decrease in unnecessary SAT solver calls the idea of guiding CEC with expressive simulation patterns is shown to be useful in verification as well.

3.8 Summary

In this chapter, we (1) present a simulation-guided logic synthesis and verification paradigm, which leverages pre-generated expressive simulation patterns to approximate the global Boolean functions with reduced need for SAT-based verification; (2) propose several strategies to generate expressive simulation patterns, including seeding with random patterns, stuck-at value checking, and resolving un-observability; (3) demonstrate the benefits of the proposed paradigm with improved resubstitution quality and reduced SAT solver calls in CEC; (4) show the reusability of the expressive patterns and counter-examples across different algorithms and with ECO modifications.

Parameters influencing the expressiveness of the simulation patterns are studied. In particular, stuck-at patterns generated with observability awareness and seeded with a small set of random patterns are found to be the most expressive. The expressive patterns are shown to be able to move runtime from optimization and verification to their pre-generation, which is advantageous because they are also shown to be reusable in resubstitution after ECO and in a different algorithm such as CEC. The experimental results show that the simulation-guided resubstitution framework allows low-cost consideration of global satisfiability don't-cares and unlimited extension of the window sizes used, which improves the average network size reduction from 1.58% to 2.77%, compared to a state-of-the-art windowing-based resubstitution algorithm. When comparing the best achievable quality of the two frameworks, a larger improvement from 3.65% to 5.83% is shown. The effectiveness of the proposed paradigm in CEC is also supported by experimental results with a 9.5% reduction in the number of SAT solver calls.

3.9 Future Work

While resubstitution guided by simulation signatures automatically accounts for satisfiability don't-cares, observability don't-cares can also be considered in resubstitution, resulting in better quality. Our preliminary result on utilizing ODCs in simulation-guided resubstitution shows about 1% further circuit size reduction at the cost of 5x more runtime.

As shown in Section 3.7.4, using expressive patterns reduces the chance of encountering counter-examples, making it possible to further reduce the use of SAT solving by validating several candidates at the same time if the majority of them are legal.

Other future works include developing strategies to refine and enhance the generated simulation patterns further and metrics to evaluate and sort the patterns. To maximize the benefit of the generated patterns, other algorithms adopting this paradigm can also be developed so that the patterns can be reused more often in a logic synthesis flow.

4 Heuristic Resynthesis

4.1 Motivation

Peephole optimization is a divide-and-conquer strategy to maintain scalability of logic synthesis algorithms, where small portions of a circuit, often referred to as *windows* or *cuts*, are extracted, optimized independently, and substituted back. With the large scale of designs nowadays, most logic synthesis algorithms, such as rewriting [MB06; MCB06; Rie+19a; RMS20], resubstitution [MB06; Mis+11b; Ama+18; Rie+18], refactoring [MB06; Ama+18; Haa+18], window rewriting [Rie+22], etc., fall into the category of peephole optimizations.

One of the important steps in any peephole optimization algorithm is re-synthesizing the extracted sub-circuit into a better one. In this work, we define the *logic resynthesis* problem as a generalized formulation of this step: The problem is given a *target* function, which is usually the root of a cut or the output(s) of a window, and some *divisor* functions, which are existing functions from neighboring nodes in the network. The resynthesis problem asks for a *dependency circuit*, computing a *dependency function*, that takes as inputs a subset of divisor functions and generates the target function at the output. If the solution is better than the original sub-network in the predefined cost metric, then it can be used to substitute the targeted node.

Various resynthesis strategies are adopted by different logic synthesis algorithms. For example, in cut rewriting, the divisor functions are always the projection (identity) functions and the target function has a small number of inputs (usually 4), thus the optimal dependency circuit can be looked up from a pre-computed database [MB06; MCB06] or be synthesized by SAT solving [Rie+19a; RMS20]. As another example, in refactoring, the divisor functions are also the projection functions, but the dependency circuit is synthesized by two-level logic optimization [MB06; Ama+18]. In contrast, in resubstitution, divisor functions other than only the projection functions are collected and used as stepping stones to construct the target function. As the number of all possible sets of divisor functions is very large, a resubstitution algorithm has to investigate the divisor functions and resynthesize the dependency circuit on the fly. Previous resubstitution works mostly attempt to enumerate small dependency circuits and compare them to the target function [MB06; Rie+18; Ama+18]. The drawback of this approach is that the dependency circuit is limited to a small size, as otherwise the search space becomes too big.

With the introduction of the simulation-guided paradigm in Chapter 3, it becomes affordable to extend the window sizes in peephole optimization. Craving for better optimization effort, resynthesis methods capable of optimizing more complex functions, which require larger dependency circuits, are in need. In a highly-optimized network where rewriting with a small cut size cannot make any further optimization, there may still be hidden optimization opportunities requiring the involvement of a larger portion of the network. In some cases, not only a larger cut (and thus a larger window) needs to be considered, but the resynthesized sub-networks should also not be limited to small ones.

4.2 Problem Formulation

4.2.1 Logic Resynthesis

Logic resynthesis (or simply *resynthesis*) is the problem of re-expressing a function in terms of other functions.

Problem Formulation 1 (*Resynthesis*)**.** Given a *target function* (or simply *target*) $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = (x_1, \ldots, x_k)$ and a collection $G = \{g_1, \ldots, g_n\}$ of n *divisor functions* (or simply *divisors*) $g_i : \mathbb{B}^k \to \mathbb{B}, 1 \le i \le n$ over the same variables, find a *dependency function* $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})), \ \forall \vec{x} \in \mathbb{B}^k. \tag{4.1}$$

■

In this formulation, variables $x_1, \ldots, x_k$ are not inputs of the function h, but any subset of them may be embedded as divisors by defining, for example, $g_1(\vec{x}) = x_1$. Also, the expression of h does not necessarily depend on all of its n inputs. In practice, a resynthesis problem may be further restricted by, for example, a set of logic operations and the number of operations allowed to be used in the expression of the dependency function. This will be further introduced in Section 4.2.2.

Example 1 (Unrestricted resynthesis)**.** Given the target function

$$f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (\neg x_2 \wedge \neg x_3) \tag{4.2}$$

and the divisor set

$$\begin{aligned} G = \{&g_1(x_1, x_2, x_3) = x_1 \wedge \neg x_2, \\ &g_2(x_1, x_2, x_3) = \neg x_2 \wedge x_3, \\ &g_3(x_1, x_2, x_3) = x_3, \\ &g_4(x_1, x_2, x_3) = x_1 \leftrightarrow x_2\}, \end{aligned} \tag{4.3}$$

one possible dependency function is

$$h(g_1, g_2, g_3, g_4) = (g_1 \vee g_4) \wedge \neg g_2. \tag{4.4}$$

Notice that Equation (4.1) is satisfied because

$$\begin{aligned} h &= ((x_1 \wedge \neg x_2) \vee (x_1 \leftrightarrow x_2)) \wedge \neg(\neg x_2 \wedge x_3) \\ &= (x_1 \wedge x_2) \vee (\neg x_2 \wedge \neg x_3) = f. \end{aligned} \tag{4.5}$$

■

The resynthesis problem can be seen as a generalization of the classical *logic synthesis* problem, where an expression or realization of h over the same variables $x_1, \ldots, x_k$ as f is sought for, i.e., G is restricted to $\{g_1 = x_1, \ldots, g_k = x_k\}$. Logic resynthesis is different from *logic decomposition* [BD97], [MSP01] or *functional decomposition* [Chu+18; LPP96], where the problem is not limited to a given divisor collection G, but involves identifying the needed divisors. In contrast, solving resynthesis problems can be seen as the core step in a *resubstitution* algorithm [MB06; Mis+11b; Ama+18; Rie+18].

4.2.2 Peephole Optimization Targeting Size Reduction

Logic resynthesis can be used in peephole optimization to optimize an extracted sub-network by resynthesizing the output function(s) of the sub-network. In this chapter, we focus on the resynthesis problem for AND-based, MAJ-based, and MUX-based circuits targeting size optimization. That is, the dependency function h is represented by an AIG, XAG, MIG, or MuxIG, called the *dependency circuit*, and the optimization goal is minimizing its size.

Example 2 (MIG resynthesis targeting size optimization)**.** Given the target function

$$f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3 \tag{4.6}$$

and the divisor set

$$
\begin{aligned}
G = \{&g_1(x_1, x_2, x_3) = x_1,\\
&g_2(x_1, x_2, x_3) = x_2,\\
&g_3(x_1, x_2, x_3) = x_3,\\
&g_4(x_1, x_2, x_3) = \mathrm{MAJ}(\neg x_1, x_2, x_3),\\
&g_5(x_1, x_2, x_3) = \mathrm{MAJ}(\neg x_1, \neg x_2, x_3)\}
\end{aligned} \tag{4.7}
$$

extracted from an MIG by a peephole optimization algorithm. The resynthesis problem is restricted to use only majority gates and inverters, and solutions with fewer gates are preferred. One possible dependency function is

$$h(g_1, g_2, g_3, g_4) = \mathrm{MAJ}(\neg g_2, g_4, \neg g_5), \tag{4.8}$$

whose corresponding dependency circuit has the least possible size of 1. ■

4.2.3 Don't-Care-Based Optimization

Most modern logic optimization algorithms place emphasis on the computation and utilization of *don't cares*, which are flexibilities in logic functions [Bar+88]. Most Boolean methods are examples of don't-care-based optimization [MCB06; RMS20; Mis+11b; Ama+18; Lee+22; Rie+22]. When solving the resynthesis problem as part of peephole optimization, it is important to take the computed don't cares into account. Although don't cares may come from different sources, namely satisfiability don't cares and observability don't cares (see Section 2.3.2), they can be treated the same when formulating the resynthesis problem.

Problem Formulation 2 (*Resynthesis with don't cares*)**.** Given a target function $f : \mathbb{B}^k \to \mathbb{B}$ over k Boolean variables $\vec{x} = (x_1, \ldots, x_k)$, a *don't-care set* $D \subseteq \mathbb{B}^k$, and a collection $G = \{g_1, \ldots, g_n\}$ of n divisor functions $g_i : \mathbb{B}^k \to \mathbb{B}, 1 \le i \le n$ over the same variables, find a dependency function $h : \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})),\ \forall \vec{x} \in \mathbb{B}^k \backslash D. \tag{4.9}$$

For convenience, we define the *care set* $C = \mathbb{B}^k \backslash D$ and the *care function* $c : \mathbb{B}^k \to \mathbb{B}$, where

$$c(\vec{x}) = \begin{cases} 1 & \vec{x} \in C, \\ 0 & \vec{x} \in D. \end{cases} \tag{4.10}$$

Thus, Equation (4.9) is equivalent to

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x})),\ \forall \vec{x} \in \mathbb{B}^k \text{ s.t. } c(\vec{x}) = 1. \tag{4.11}$$

Moreover, if we define the target *onset function* $f_{\mathrm{on}} = f \wedge c$ and the *offset function* $f_{\mathrm{off}} = \neg f \wedge c$, then Equation (4.9) is also equivalent to

$$h\Big(f_{g_1}(\vec{x}),\dots,f_{g_n}(\vec{x})\Big) \implies \neg f_{\text{off}}(\vec{x}) \text{ and } f_{\text{on}}(\vec{x}) \implies h\Big(f_{g_1}(\vec{x}),\dots,f_{g_n}(\vec{x})\Big), \forall \vec{x} \in \mathbb{B}^k. \quad (4.12)$$

■

Example 3 (Resynthesis with nonempty don't-care set)**.** Suppose we have the same target function f and divisor set G as in Example 1 (Equations (4.2) and (4.3), respectively). Additionally, we are now given the care function

$$c(x_1,x_2,x_3) = x_2 \vee (x_1 \leftrightarrow x_3).$$

In other words, the don't-care set $D = \{(1,0,0),(0,0,1)\}$ is nonempty. For this relaxed problem, one possible dependency function is

$$h(g_1,g_2,g_3,g_4) = g_4, \quad (4.13)$$

which is simpler than Equation (4.4) thanks to the provided don't cares. Notice that Equation (4.9) is satisfied because the difference between f and h ($f \oplus h = \{(1,0,0),(0,0,1)\}$) does not intersect with the care set. ■

4.2.4 Simulation-Guided Logic Synthesis

The resynthesis algorithms proposed in this chapter are compatible with the simulation-guided paradigm described in Chapter 3. In this case, the target and divisor functions are represented by the simulation signatures of the corresponding nodes in the network and *partial truth tables* are used as the data structure. A partial truth table is a truth table of arbitrary length l, representing a partially-specified, incomplete function $f : X \to \mathbb{B}$, where $X \subseteq \mathbb{B}^k$ and k is the number of primary inputs of the network. The i-th bit $T[f]_i$ is the output of f under the i-th simulation pattern in the set. What patterns are used in simulation is not important for the resynthesis problem. It is only required that the partial truth tables of the target and divisors are simulated using the same ordered set of simulation patterns.

Problem Formulation 3 (*Resynthesis with incompletely-specified functions*)**.** Given a target function $f : X \to \mathbb{B}$ and a collection $G = \{g_1,\dots,g_n\}$ of n divisor functions $g_i : X \to \mathbb{B}, 1 \le i \le n$ defined over the same input space $X \subseteq \mathbb{B}^k, k \in \mathbb{N}^+$, find a dependency function $h :\ \mathbb{B}^n \to \mathbb{B}$ satisfying

$$f(\vec{x}) = h(g_1(\vec{x}),\dots,g_n(\vec{x})),\ \forall \vec{x} \in X. \quad (4.14)$$

Optionally and similarly to the problem formulation in Section 4.2.3, a don't-care set $D \subseteq X$ may be given. The care set is then $C = X \backslash D$, and the care function $c : X \to \mathbb{B}$ is defined the same as in Equation (4.18). ■

A resynthesis algorithm receiving target and divisor functions as truth tables does not distinguish the case where functions are incompletely-specified from where they are completely-specified. A solution given by the algorithm fulfills Equation (4.14), and it is up to the simulation-guided framework to validate the dependency circuit in the context of the network and add more bits into the partial truth tables to block invalid solutions.

4.3 Overview

In this chapter, we propose three heuristic resynthesis algorithms to be used in peephole optimizations of, respectively, AIGs, MIGs, and MuxIGs. The proposed resynthesis algorithms share the following characteristics:

- Support for incomplete functions and don't cares: The divisor and target functions may be given as completely-specified Boolean functions or partial simulation signatures [Lee+22]. The algorithms resynthesize dependency circuits satisfying the given parts of functions and make no assumption on the uninformed parts. Moreover, don't cares of the target function may be given, and the algorithms take advantage of this information to resynthesize smaller dependency circuits.
- Heuristic but unlimited: Optimality may only be guaranteed when the optimal solution is small. It is also not guaranteed that a solution is always found. Nevertheless, there is no limit on the possible solution size. When a small-sized solution does not exist, the heuristic may still find a larger solution that exact methods can never find within reasonable runtime.
- Top-down decomposition: Although the three proposed algorithms are designed differently, they all start from choosing "good" divisors based on some evaluation criteria involving the target function. Then, if the target cannot be realized within a few gates, it is decomposed into easier-to-realize targets by a gate on top.

4.4 Related Works

In this section, we introduce previous works dealing with the same or similar problems.

4.4.1 Functional Dependency by Interpolation

In [Lee+07], a method to find *functional dependency* using interpolation was proposed. The problem of finding functional dependency is essentially the same as the unrestricted logic resynthesis problem (Problem Formulation 1), where the goal is only to find a dependency function without a particular focus on (minimizing) the corresponding dependency circuit. In [Lee+07], given a target function f and a set of base functions G (i.e., divisor functions in our terminology), it is first checked if f functionally depends on G, i.e., if a dependency function h exists. This is done by solving a SAT problem consisting of two copies of the circuit representation of f and G and additional constraints that the outputs of G are the same, but one copy outputs $f = 0$ and the other outputs $f = 1$. Intuitively, the SAT problem encodes that there exists a pair of offset $\vec{x}_0$ and onset $\vec{x}_1$ minterms of f, such that $g_i(\vec{x}_0) = g_i(\vec{x}_1)$ for all $g_i \in G$. A dependency function h exists if and only if the SAT instance is unsatisfiable, and such h can be computed by deriving the interpolant from the refutation proof given by the SAT solver.

The interpolation-based method was later used in [Mis+11b] as part of resubstitution for k-LUT networks. Because the dependency function is implemented as a LUT node, it is not needed to construct a dependency circuit. However, for resubstitution algorithms for AIGs, XAGs, or MIGs, etc., the size of the dependency circuit is crucial for the optimization quality. Thus, the interpolation-based method is not applicable there. Also, as the procedure involves constructing CNF clauses of a circuit computing f and G, it cannot solve the resynthesis problem with incomplete simulation signatures (Problem Formulation 3).

4.4.2 SAT-Based Exact Synthesis

SAT solving can also be used to find the *smallest* dependency circuit, instead of just *some* feasible dependency function. SAT-based exact synthesis of Boolean chains encodes the following question into a CNF formula: "Does there exist a Boolean chain which implements the given function f with exactly r steps[1]?" A solution Boolean chain can be interpreted from a satisfiable assignment to the encoded CNF formula, whereas an unsatisfiable result means a solution of r steps is impossible. By solving such SAT problem iteratively with different values of r, the smallest feasible r can be found [Knu11]. While SAT-based exact synthesis was originally described to synthesize a Boolean chain computing a given function at its output(s) in terms of its input variables, i.e., it solves a subset of the resynthesis problem where divisors are projection functions, it can be modified and extended to solve the general resynthesis problem where divisors can be any functions and don't cares are supported [RMS20]. In [Haa+20], different CNF encodings of the problem were analyzed and compared. However, although it is possible to reduce the number of variables involved in the SAT instance, it is done at the cost of more clauses in the CNF formula. As the intrinsic complexity of the problem is exponential, the scalability of an exact algorithm is always limited.

4.4.3 Enumeration-Based Resubstitution

Resubstitution is a logic optimization technique which substitutes a node in the network with another existing node, or with newly-created nodes constructed upon other existing nodes [Bra+87]. Resubstitution for AIG size minimization was first proposed in [MB06], where windows of no more than 16 inputs are constructed to collect structurally-proximate divisor

[1]Using the terminology in this thesis, a Boolean chain with r steps is a logic network with r nodes, where each node models an arbitrary logic gate. Additional clauses may be added to the CNF formula to constrain possible gate types to a predefined set.

nodes and to perform complete local simulation. Small sub-networks of up to three AND gates and taking divisors as inputs are enumerated, simulated and compared to the target function. If the composed function is the same as (or compatible subject to the care set) the target, a viable dependency circuit is found. Such search for resubstitutions is essentially the AIG resynthesis problem with size awareness. The complexity of the enumeration-based resynthesis approach is $\mathcal{O}(|G|^{|H|+1})$, where $|G|$ is the number of divisors and $|H|$ is the size of possible dependency circuit. Thus, $|G|$ is limited to at most 150 and $|H|$ is limited to at most 2 in [MB06].

In [Ama+18], enumeration-based resynthesis was extended to larger dependency circuits, but still limited to some predefined structures such as AND-XOR, MUX, MUX-XOR, etc. A Boolean filtering rule was proposed to filter out useless divisors, so that the search space was reduced. Overall, eight types of dependency circuit structures are tried in the increasing order of their size, and for each structure, filtered set of divisors are enumerated at the inputs similarly to [MB06].

An enumeration-based resubstitution for MIGs was first proposed in [Rie+18]. The algorithm enumerates dependency circuits of up to two MAJ gates. Two efficiency enhancement techniques were proposed: (1) A filtering rule derived from the majority law is applied:

$$\text{if } x \neq y \text{ and } \exists z, \text{MAJ}(x, y, z) = f, \text{ then MAJ}(x, y, f) = f \tag{4.15}$$

(2) As a preprocessing step, the truth tables are normalized to have the first bit always 1, such that the number of inversion cases to investigate is reduced. Truth tables having a 0 as the first bit are complemented and the inversion is recorded.

In addition to enumerating small dependency circuits, a special type of node replacement, called *R-resubstitution*, is explored. R-resubstitution exploits the *relevance rule* of majority gates [AGD16]:

$$\text{MAJ}(x, y, z) = \text{MAJ}(x_{y/\bar{z}}, y, z), \tag{4.16}$$

where $x_{y/\bar{z}}$ is obtained by replacing all occurrences of y with $\neg z$ in x. Instead of substituting the root node with a dependency circuit in the classical resubstitution, R-resubstitution substitutes a fanin node x of the root $r = \text{MAJ}(x, y, z)$ with a divisor d if $(x \oplus d)(y \oplus z) = 0$ and r is the only fanout of x. Unfortunately, finding R-resubstitution cannot be formulated as a resynthesis problem, thus it is not considered in the rest of this chapter.

The core problem resubstitution algorithms solve is logic resynthesis. Existing works on resubstitution are based on enumeration, thus there exist small upper bounds on the size of dependency circuits they can find. In contrast, the heuristic resynthesis algorithms proposed in this work are unlimited in this respect.

4.4.4 Akers' Majority Synthesis

Akers' majority synthesis algorithm was the earliest work on heuristic synthesis of MIGs [Ake62]. It is a bottom-up approach that builds new gates using the constructed ones. In [Ake62], Akers' Algorithm was presented to synthesize an MIG for any given function from primary inputs, but the algorithm can actually also solve the MIG resynthesis problem. First, the truth tables of the primary inputs are *normalized* by taking their XNOR with the target function, such that the goal of the algorithm becomes building the constant 1 function. The main data structure in Akers' Algorithm, called the *unitized table*, is a collection of the normalized truth tables of primary inputs (and their negations) and of the outputs of MAJ gates created throughout the algorithm. Each column of the unitized table corresponds to a node (a PI or a gate) that can be used to build the next gate, and each row corresponds to a value assignment to the PIs (i.e., a minterm). The algorithm iteratively *reduces* the unitized table, by removing redundant columns and dominated rows, and *expands* the unitized table, by choosing three columns to build a new MAJ gate and adding a new column. The procedure repeats until there is only one column of all 1s left, or until the resource limit exceeds. The choice on using which columns to build new gates is heuristic, so the algorithm does not guarantee to always find a solution.

4.5 Heuristic AND-Based Resynthesis

In this section, we introduce the heuristic AND-based resynthesis algorithm which resynthesizes an AIG or an XAG. The algorithm primarily considers AND gates (and cost-free inverters), but it may be extended to consider XOR gates as well, although in a limited way. The algorithm is based on (a) classification of divisors and (b) recursive decomposition. The former idea has been practically adopted in enumeration-based resubstitution [MB06], but rarely described in the literature. In Section 4.5.1, we give the definition of the unateness of divisors and explain why it is useful in reducing the search space of resynthesis. On top of that, in Section 4.5.3, we propose the recursive decomposition, which is key for our resynthesis algorithm being unbounded by the solution size.

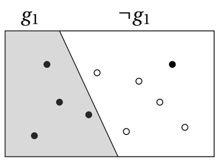
We use figures to illustrate essential concepts in this section. In Figures 4.1 to 4.3, a rectangle marks the Boolean space under which the target and divisor functions are defined ($\mathbb{B}^k$ in Problem Formulations 1 and 2 or X in Problem Formulation 3). Black dots in the rectangle represent onset minterms of the target and white dots represent offset minterms. In the space where no dots are present, there can be don't-care minterms. For clearer illustration, don't-care minterms are plotted as gray dashed dots in Figure 4.3.

A divisor function g separates the Boolean space into two halves, the region where $g = 1$ and the region where $g = 0$ (or equivalently, $\neg g = 1$). We refer to a divisor with or without negation as a *literal*, i.e., a literal is either a divisor g or a negated divisor $\neg g$, corresponding respectively to the two halves of the Boolean space.

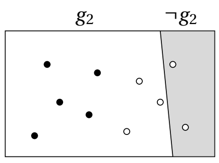
4.5.1 Classification of Divisors

Any composition of some divisor functions is also a function defined over the same Boolean space, thus also separates the space into two halves. For example, composing two literals l_1, l_2 with an AND gate results in a separation where the region $l_1 \wedge l_2 = 1$ is the intersection of the regions $l_1 = 1$ and $l_2 = 1$, and the region $l_1 \wedge l_2 = 0$ is the union of the regions $l_1 = 0$ and $l_2 = 0$. The goal of the resynthesis algorithm is to find a composition whose resulting function separates the Boolean space into a half containing only onset minterms of the target and a half containing only offset minterms.

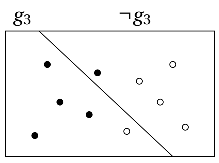
We observe that, if two literals l_1, l_2 are to be composed using an AND gate and realizing the target, then the regions $l_1 = 0$ and $l_2 = 0$ must not contain any onset minterm of the target. Similarly, if two literals l_3, l_4 are to be composed using an OR gate (equivalent to an AND gate with input and output negations) and realizing the target, then the regions $l_3 = 1$ and $l_4 = 1$ must not contain any offset minterm of the target because the resulting region $l_3 \vee l_4 = 1$ is the union of the regions $l_3 = 1$ and $l_4 = 1$. We call such property *unateness.*



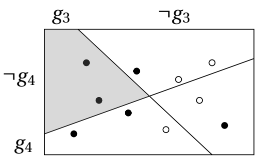
(a) Literal g_1 is positive unate.



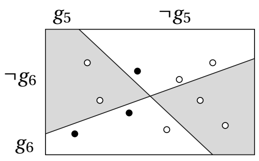
(b) Literal $\neg g_2$ is negative unate.



(c) g_3 is a binate divisor.



(d) AND-pair $g_3 \wedge \neg g_4$ is positive unate.



(e) XOR-pair $g_5 \oplus g_6$ is negative unate.

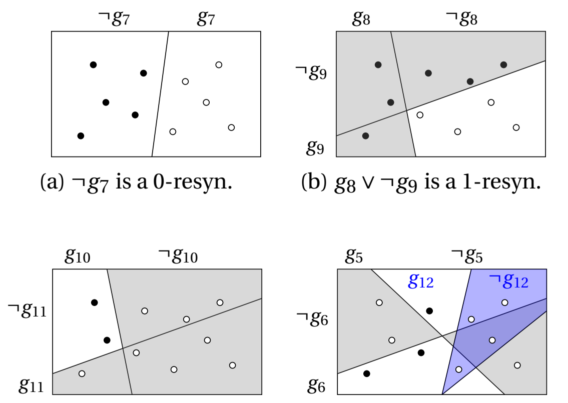
Figure 4.1: Illustration of unate literals and binate divisors.

A literal l is said to be *positive unate* if $l \wedge f_{\text{off}} = 0$. For example, in Figure 4.1 (a), g_1 is positive unate. Similarly, a literal l is said to be *negative unate* if $l \wedge f_{\text{on}} = 0$. For example, in Figure 4.1 (b), $\neg g_2$ is negative unate. In contrast to unate literals, binateness is defined for divisors. Given a divisor g, if both g and $\neg g$ are neither positive nor negative unate, then g is said to be a *binate* divisor. For example, in Figure 4.1 (c), g_3 is a binate divisor. Note that unateness is defined for literals and binateness is defined for divisors. A (non-binate) divisor g may have one of its literals being unate, but the other literal being neither positive nor negative unate, such as g_1 in Figure 4.1 (a) and g_2 in Figure 4.1 (b). Also note that these definitions are different from the unateness of a Boolean function with respect to a variable [McN61].

Only unate literals can be used to construct the target function using one gate. Thus, by classifying divisors, the number of comparisons required to identify dependency circuits of no more than one gate is reduced. Nevertheless, binate divisors are not totally useless. Two binate divisors may be composed with a gate and become unate. Thus, the definitions of positive and negative unateness are extended for pairs of literals. A pair p of two literals l_1, l_2 obtained from (optionally negating) two binate divisors is said to be a positive unate *AND-pair* if $(l_1 \wedge l_2) \wedge f_{\text{off}} = 0$. For example, in Figure 4.1 (d), $(g_3, \neg g_4)$ is a positive unate AND-pair. Similarly, it is negative unate if $(l_1 \wedge l_2) \wedge f_{\text{on}} = 0$. When finding unate pairs, we investigate all pairs of two binate divisors and all of the four possible inverter configurations, corresponding to the four regions of the Boolean space divided by the two divisor functions. There is no need to try an OR-pair because composing two binate divisors with an OR gate (i.e., taking the union) will never lead to a unate function. If XOR gates are allowed, we additionally try to find unate XOR-pairs. For example, in Figure 4.1 (e), (g_5, g_6) is a negative unate XOR-pair.

Algorithm 4.1: Heuristic AND-based resynthesis algorithm.

```
Input: target onset $f_{on}$, target offset $f_{off}$, divisors $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit $H$
if $f_{on} = 0$ then return Constant 0
if $f_{off} = 0$ then return Constant 1

$U_p \leftarrow positive\_unate(G, f_{off})$
$U_n \leftarrow negative\_unate(G, f_{on})$
$B \leftarrow binate(G, U_p, U_n)$

if $u \leftarrow find\_0resyn(U_p, U_n)$ then return $u$

$U_p \leftarrow sort(U_p, f_{on})$
$U_n \leftarrow sort(U_n, f_{off})$

if $u, v \leftarrow find\_1resyn(U_p, f_{on})$ then return $u \vee v$
if $u, v \leftarrow find\_1resyn(U_n, f_{off})$ then return $\neg u \wedge \neg v$

$P_p \leftarrow positive\_unate\_pair(B, f_{off})$; $P_p \leftarrow sort(P_p, f_{on})$
$P_n \leftarrow negative\_unate\_pair(B, f_{on})$; $P_n \leftarrow sort(P_n, f_{off})$

if $p, u \leftarrow find\_2resyn(P_p, U_p, f_{on})$ then return $(p_1 \circ_p p_2) \vee u$
if $p, u \leftarrow find\_2resyn(P_n, U_n, f_{off})$ then return $\neg(p_1 \circ_p p_2) \wedge \neg u$
if $p, q \leftarrow find\_3resyn(P_p, f_{on})$ then return $(p_1 \circ_p p_2) \vee (q_1 \circ_q q_2)$
if $p, q \leftarrow find\_3resyn(P_n, f_{off})$ then return $\neg(p_1 \circ_p p_2) \wedge \neg(q_1 \circ_q q_2)$

$u \leftarrow choose\_top(U_p, U_n, P_p, P_n)$
$f'_{on} \leftarrow new\_target(u, f_{on})$
$f'_{off} \leftarrow new\_target(u, f_{off})$
$H_r \leftarrow resynthesize(f'_{on}, f'_{off}, G)$
return $u \circ_u H_r$
```



(a) $\neg g_7$ is a 0-resyn. (b) $g_8 \vee \neg g_9$ is a 1-resyn.

(c) $g_{10} \wedge \neg g_{11}$ is a 1-resyn. (d) $g_{12} \wedge \neg(g_5 \oplus g_6)$ is a 2-resyn.

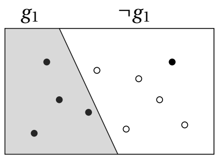
Figure 4.2: Illustration of composing simple dependency circuits.

4.5.2 Simple Dependency Circuits

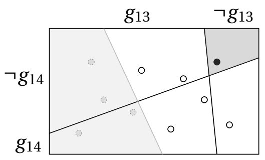
Simple dependency circuits of no more than three gates are identified similarly to the enumeration-based method. First, if the target onset or offset is empty, then it can be realized with a constant (lines 1-2 in Algorithm 4.1). After classifying divisors and collecting unate literals as described in Section 4.5.1 (lines 4-6), we first check if there exists a literal that realizes the target without extra gates. That is, if a literal l is positive unate and its negation $\neg l$ is negative unate, then l realizes the target (line 8). We call this a 0-*resyn* because it has 0 gates in the dependency circuit. For example, in Figure 4.2 (a), $\neg g_7$ is positive unate and g_7 is negative unate, thus $\neg g_7$ is a 0-resyn.

To find dependency circuits with one gate, called 1-*resyn*, we try to compose two positive unate literals with an OR gate, or to compose two negative unate literals with an AND gate (lines 13-14). For each pair l_1, l_2 of positive unate literals, we check if their union contains all of the onset minterms. That is, if $\neg(l_1 \vee l_2) \wedge f_{\text{on}} = 0$, or equivalently, $\neg l_1 \wedge \neg l_2 \wedge f_{\text{on}} = 0$. We do not need to check for offset minterms thanks to the definition of positive unate literals. For example, Figure 4.2 (b) is an OR-type 1-resyn because there is no more onset minterms in the white region. Similarly, two negative unate literals l_3, l_4 form an AND-type 1-resyn if their union contains all of the offset minterms. That is, $\neg l_3 \wedge \neg l_4$ realizes the target if $\neg l_3 \wedge \neg l_4 \wedge f_{\text{off}} = 0$, such as Figure 4.2 (c). As the condition to be checked in this step is whether the union of two literals contains all onset (for positive unate) or offset (for negative unate) minterms, we first sort the literals based on how many onset or offset minterms they contain (lines 10-11). This way, we may terminate the investigation earlier when we know the remaining pairs of literals all have a total number of onset (or offset) minterms less than the number of onset (or offset) minterms of the target.

If a dependency circuit of size no more than one cannot be found, we proceed to collect unate



(a) Decompose f_{on} with a positive unate literal g_1.



(b) f'_{on} can be more easily realized by $\neg g_{13} \wedge \neg g_{14}$.

Figure 4.3: Illustration of the recursive decomposition.

pairs (lines 16-17) and try to find a 2-*resyn* (lines 19-20) or 3-*resyn* (lines 21-22). A 2-resyn is composed of a unate literal and a unate pair. The conditions to be checked are similar to those for 1-resyn. For example, in Figure 4.2 (d), a negative unate literal $\neg g_{12}$ and a negative unate XOR-pair (g_5, g_6) (taken from Figure 4.1 (e)) forms an AND-type 2-resyn. Similarly, a 3-resyn is composed of two unate pairs. In Algorithm 4.1, we use $\circ$ to denote an unspecified gate type depending on the pair noted as the subscript, and we use p_1, p_2 to denote the two elements of a pair p.

4.5.3 Recursive Decomposition

When the target cannot be realized within three gates, the algorithm heuristically chooses an unate literal or an unate pair to decompose the target function (lines 24-28). If a positive unate literal l_1 is chosen, a new target onset $f'_{\text{on}} = f_{\text{on}} \wedge \neg l_1$ with fewer minterms is derived by constructing the dependency circuit with an OR gate on top, having l_1 as one of its fanins. Then, Algorithm 4.1 is recursively called on the new onset f'_{on} and the same offset $f'_{\text{off}} = f_{\text{off}}$ (line 27) to construct the remaining circuit as the other fanin of the top OR gate. For example, in Figure 4.3 (a), we decompose f_{on} with a positive unate literal g_1 (taken from Figure 4.1 (a)), resulting in f'_{on} in Figure 4.3 (b). The new f'_{on} has only one onset minterm remaining and is more easily realized by $\neg g_{13} \wedge \neg g_{14}$, which were both binate before decomposition. The original target function is thus realized by $g_1 \vee (\neg g_{13} \wedge \neg g_{14})$.[2] In contrast, if a negative unate literal l_2 is chosen, the target onset stays the same, whereas a new offset $f'_{\text{on}} = f_{\text{off}} \wedge \neg l_2$ is derived. The dependency circuit is then constructed with an AND gate with negated fanins on top.

The choice on which literal or pair to use to decompose (line 24) is made by comparing the number of onset (for positive unate literals or pairs) or offset (for negative unate) minterms they contain. The one containing the most minterms is preferred. However, a pair is only chosen if it contains more than twice the number of minterms than the winning literal because choosing a pair leads to one more gate in the dependency circuit.

[2]The example is made simple for easier understanding. This solution can actually be found as a 2-resyn without the recursive decomposition. To give a real example where recursive decomposition is needed, for example, g_{13} and g_{14} could be pairs instead of divisors, which only become unate with respect to the new onset f'_{on}.

4.5.4 Summary of AND-Based Resynthesis

Algorithm 4.1 summarizes the AND-based resynthesis algorithm. In Algorithm 4.1, lines 1-26 are similar to enumeration-based resubstitution, which resynthesizes dependency circuits of at most 3 gates. Lines 24-28 are the key for the algorithm to resynthesize larger dependency circuits, where line 27 calls the resynthesis algorithm recursively.

It is neglected in the pseudocode, but in practice an additional parameter *size limit* is passed to the algorithm. Before each step, the size limit is checked and the algorithm terminates without a solution if the limit is reached. For example, before *find_3resyn*, if *size limit* is 2, the algorithm returns *no solution*. In line 27, the *size limit* being passed to the recursive call is the current size limit minus 1 (when decomposing with a literal) or 2 (when decomposing with a pair). When the algorithm returns *no solution*, it is possible that a solution larger than *size limit* exists and can be found if *size limit* were set larger, or that the given problem is infeasible. It is also possible that a solution exists, but cannot be found by the algorithm because it is heuristic, irrelevant to *size limit*. The same early-termination mechanism also applies to the following MAJ-based and MUX-based resynthesis algorithms.

4.6 Heuristic MAJ-Based Resynthesis

We introduce the heuristic MAJ-based resynthesis algorithm in this section, based on the following key ideas:

- *Normalization*: Divisor functions are normalized to simplify the algorithm and reduce the number of bitwise operations needed. This step is done only once in the beginning. (Section 4.6.1)

- *Covering the care function*: We introduce the notion of *care functions* at any position in the dependency circuit under construction. The goal of the algorithm is to *cover* more uncovered bits in the care function by modifying the current dependency circuit until all bits are covered. (Section 4.6.2)

- *Heuristic choice of divisors*: The algorithm repeatedly chooses three divisors to form a new majority gate. Divisors are chosen according to their evaluation on a heuristic weight function with respect to the current care function. (Section 4.6.3)

- *Expansion to a tree-like circuit*: The algorithm constructs the dependency circuit by repeatedly expanding on a leaf of the circuit. It chooses a fanin of a gate which is connected to a divisor, takes out the divisor, and replaces it with a newly-constructed gate. The resulting circuits thus have tree-like structures. (Section 4.6.4)

4.6.1 Normalization

Given the target f and the set of divisors $G = \{g_1, \dots, g_n\}$, the divisors are *normalized* by computing their XNOR with the target. By doing so, the logic of the algorithm is simplified—comparing the output function of the dependency circuit against the target simplifies to testing if the output function is a tautology. Moreover, due to the self-duality property of the majority function [MTT61], inverters can always be pushed to the primary inputs. Hence, we limit our search to dependency circuits without internal inverters and consider inverters only at the inputs by supplementing the divisor set with negated literals. The set N of normalized literals to be chosen from as inputs to the dependency circuit is computed by

$$N = \{l_{2i-1} = g_i \leftrightarrow f,\ l_{2i} = \neg g_i \leftrightarrow f \mid 1 \le i \le n\}. \tag{4.17}$$

4.6.2 The Care Function

Consider a MAJ gate with function $y = \mathrm{MAJ}(x_1, x_2, x_3)$ and a certain bit position p in its truth table. In order to have $T[y]_p = 1$, we must have

$$T[x_i]_p = T[x_j]_p = 1, \text{ where } i, j \in \{1, 2, 3\} \text{ and } i \neq j.$$

If the functions x_1 and x_2 have been decided but x_3 is still flexible, then we require $T[x_3]_p = 1$ only if $T[x_1]_p = 0$ or $T[x_2]_p = 0$. In such case, we say that p is a *care bit* for the third fanin of the gate under construction.

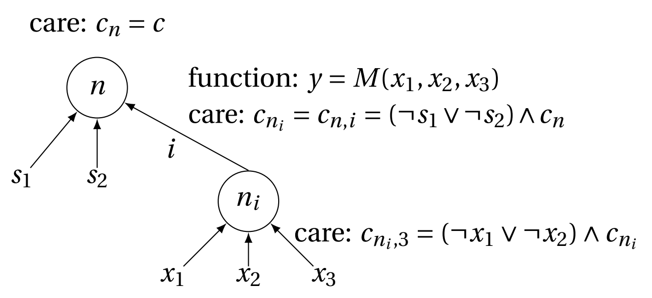

care: $c_n = c$
n
function: $y = M(x_1, x_2, x_3)$
care: $c_{n_i} = c_{n,i} = (\neg s_1 \lor \neg s_2) \land c_n$
i
s_1
s_2
n_i
care: $c_{n_i,3} = (\neg x_1 \lor \neg x_2) \land c_{n_i}$
x_1
x_2
x_3

Figure 4.4: Illustration of the care functions.

Generalizing and extending to all bit positions, we define the *care function* c_i of a fanin i to a node n as

$$c_{n,i} = (\neg s_1 \lor \neg s_2) \land c_n, \tag{4.18}$$

where s_1 and s_2 are the other two fanin functions of n (i.e., siblings of i) and c_n is the care function of n. If n is the topmost node of the dependency circuit, as in Figure 4.4, then its care function c_n is the care function c of the target, given as input to the resynthesis problem. Otherwise, as our dependency circuits are tree-like, the node n must have exactly one fanout (parent) node, and its care function is derived using Equation (4.18) according to its parent's care function and its siblings' functions. For example, the care function c_{n_i} of node n_i in Figure 4.4 is the care function of the fanin i to node n.

A care bit in a care function is said to be *covered* if the function presented at the node (or at the fanin edge) indeed provides 1 at this bit. For example, for a care bit in $c_{n_i,3}$ to be covered, the function x_3 needs to be 1 at this bit. If the care function of a node (for example, c_{n_i} in Figure 4.4) is of interest, then we need at least two fanin functions of the node (for example, x_1 and x_3) to *cover* the bit by having 1's.

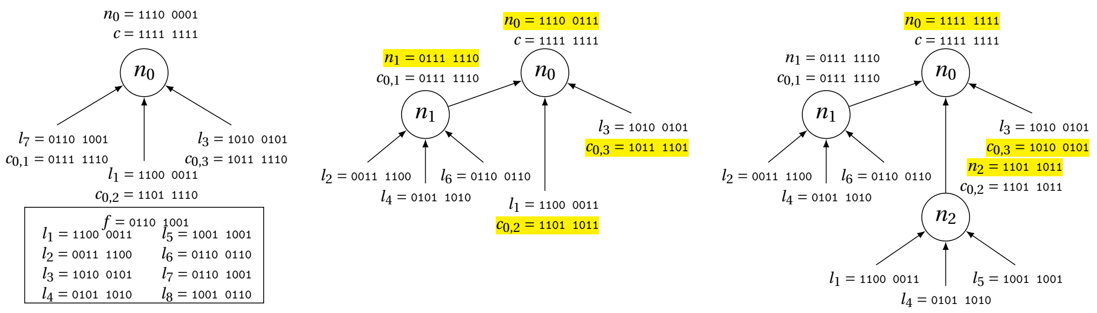
4.6.3 Choosing Divisors

Given the care function c_n of a node n, a heuristic selection is used to choose three literals l_1, l_2, l_3 from N to construct a MAJ gate, aiming at maximizing $\textsc{Ones}(\text{MAJ}(l_1, l_2, l_3) \wedge c_n)$:

$$\begin{aligned}
&l_1 = \underset{l \in N}{\operatorname{argmax}}(\textsc{Ones}(l \wedge c_n)) \\
&l_2 = \underset{l \in N_2}{\operatorname{argmax}}(\textsc{Ones}(l_1 \wedge l \wedge c_n) + 2 \cdot \textsc{Ones}(\neg l_1 \wedge l \wedge c_n)) \\
&l_3 = \underset{l \in N_3}{\operatorname{argmax}}(\textsc{Ones}((l_1 \oplus l_2) \wedge l \wedge c_n) + 2 \cdot \textsc{Ones}((\neg l_1 \wedge \neg l_2) \wedge l \wedge c_n) \\
&\text{where } N_2 = N \backslash \{l_1, \neg l_1\}, N_3 = N_2 \backslash \{l_2, \neg l_2\}
\end{aligned} \tag{4.19}$$

The first literal is chosen to cover most care bits. When choosing the second literal, the care bits covered by the first literal still need to be covered again, thus we acknowledge more $\textsc{Ones}(l_1 \wedge l \wedge c_n)$. But more importantly, we are more eager to cover the care bits that are not covered by the first literal, thus the weight for $\textsc{Ones}(\neg l_1 \wedge l \wedge c_n)$ is doubled. For the last literal, the care bits that are already covered twice can be ignored; the care bits covered only once $((l_1 \oplus l_2) \wedge l \wedge c_n)$ seek to be covered again; the care bits that are never covered before $((\neg l_1 \wedge \neg l_2) \wedge l \wedge c_n)$ appear to be more difficult to cover than the other bits and they are thus doubly weighed. In the last case, it may seem counter-intuitive to cover these bits with the last literal because covering them only once is not enough. However, the first two literals may be replaced by new nodes later on in the algorithm, so it is still useful to cover them at least once in this stage.

This evaluation step will be repeatedly incurred throughout the algorithm. The computational complexity is linear to the number of divisors, which can be large. We observe that the resulting choice depends solely on the care function c_n. To speed up the computation, a computed table can be used to cache the results. This is implemented as a hash table mapping from a care function to three divisors.



(a) The topmost node n_0.

(b) Expand at $(n_0, 1)$ with $n_1 = \mathrm{MAJ}(l_2, l_4, l_6)$.

(c) Expand at $(n_0, 2)$ with $n_2 = \mathrm{MAJ}(l_1, l_4, l_5)$.

Figure 4.5: Example of MAJ-based resynthesis.

4.6.4 Expansion

When all care bits of the three fanins of the topmost node are covered, the constant 1 function is successfully derived at its output and the algorithm terminates. After constructing the first node with three literals, we choose one of the fanins with uncovered care bits, if any, and try to cover more care bits by replacing the literal with a new gate. This process is called an *expansion*.

To *expand* a fanin, the original literal is temporarily taken away. Then, three literals are chosen as the fanins of the new gate using Equation (4.19). After an expansion, the function at the expanded fanin is different, thus the functions of its transitive fanouts, as well as the care functions of its siblings, are updated accordingly. Until the constant 1 is derived at the output of the topmost node by covering all the care bits, the algorithm proceeds by choosing another position to expand. An *expansion position* is a fanin of any node which is connected to a literal and whose care function is not fully covered. Heuristically, we choose the position with the least uncovered care bits to be expanded first because it is closest to be fully covered.

It is possible that the majority output of the three chosen literals does not cover more care bits than the original literal. Hence, the new gate is only constructed and used to replace the original literal if the number of covered care bits increases. When an expansion position is tried but the coverage of care bits does not increase, the new gate is discarded and the position is marked as visited to avoid trying it again. However, if its care function is updated because of an update in the function of one of its siblings, the visited flag is reset and the expansion position may be tried again. To avoid constructing gates using the same literals repeatedly as a chain, when the care function of a node is the same as one of its fanins, the expansion position at this fanin is directly marked as visited without trying to expand it.

Algorithm 4.2: Heuristic MAJ-based resynthesis algorithm.

```
Input: target function $f$, care function $c$, divisor functions $G = \{g_1, \ldots, g_n\}$
Output: dependency circuit $H$
1  $N \leftarrow$ normalize$(G, f)$
2  $n_0 \leftarrow$ choose_literals$(N, c)$
3  $H \leftarrow \{n_0\}$
4  while $n_0$.output $\neq 1$ do
5      $(n_p, i) \leftarrow$ choose_expansion_position$(H)$
6      $n \leftarrow$ choose_literals$(N,\ n_p$.fanin$(i)$.care$)$
7      if accept_expansion$(n_p, i, n)$ then
8          $n_p$.fanin$(i) \leftarrow n$
9          update$(H)$
10     else
11         mark_visited$(n_p, i)$
12 return $H$
```

4.6.5 Summary and Example of MAJ-Based Resynthesis

Algorithm 4.2 summarizes the heuristic MAJ-based resynthesis algorithm. First, the set of divisors is normalized and supplemented using Equation (4.17) (line 1). Then, the top node n_0 is constructed by choosing three literals using Equation (4.19) and added into the dependency circuit as the first node (lines 2-3). If the output function of n_0 is not constant 1 (line 4), we choose an expansion position (the i-th fanin of a parent node n_p) which is currently connected to a literal (line 5). The care function of the position is computed by Equation (4.18) and used to choose three literals to construct a new gate (line 6). If replacing the original literal with the new gate increases the number of covered care bits, the expansion is accepted and the dependency circuit is updated (lines 7-9); otherwise, the position is marked as visited (lines 10-11). The expansion procedure is repeated until the constant 1 function is obtained at the output of the topmost node.

An example execution of the algorithm is illustrated in Figure 4.5, where the target function is

$$f(\vec{x}) = x_1 \oplus x_2 \oplus x_3, \tag{4.20}$$

the care function $c = 1$, and the set G of divisors consists of

$$G = \{g_1(\vec{x}) = x_1, g_2(\vec{x}) = x_2, g_3(\vec{x}) = x_3, g_4(\vec{x}) = 0\}. \tag{4.21}$$

The normalized set N of literals, computed according to Equation (4.17), is listed in their truth table representations in the box in Figure 4.5 (a). The yellow-shaded parts Figure 4.5 are the truth tables being updated after expansions. First, in Figure 4.5 (a), given the care function $c = 1$, three literals l_7, l_1, l_3 are chosen according to Equation (4.19) to form the topmost node n_0, computing the function at its output $n_0 = \text{MAJ}(l_7, l_1, l_3)$. Care functions of each fanin $c_{0,i}$ are computed according to Equation (4.18). Then, in Figure 4.5 (b), the first fanin of n_0 is chosen to be expanded with a new node n_1. According to its care function $c_{0,1}$, three literals l_2, l_4, l_6 are chosen. The function at the expanded fanin is updated with $n_1 = \text{MAJ}(l_2, l_4, l_6)$. Following which, the care functions at its siblings $c_{0,2}$ and $c_{0,3}$, as well as the output function n_0 are also updated. After the expansion, all care bits of the first fanin of n_0 have been covered by the function of n_1, but there are still two care bits in each of the updated $c_{0,2}$ and $c_{0,3}$ not yet covered. So, in Figure 4.5 (c), the second fanin of n_0 is expanded with another new node n_2. Similarly, according to its care function $c_{0,2}$, three literals l_1, l_4, l_5 are chosen, and the node functions n_2 and n_0, as well as the sibling's care function $c_{0,3}$, are updated. Now, all care bits in $c_{0,2}$ and also $c_{0,3}$ are covered, and the output function of n_0 is constant 1. The resynthesis has thus been completed. The final solution is $h(g_1, g_2, g_3, g_4) = \text{MAJ}(\text{MAJ}(\neg g_1, \neg g_2, \neg g_3), \text{MAJ}(g_1, \neg g_2, g_3), g_2)$.

4.7 Heuristic MUX-Based Resynthesis

Although rarely researched on, MuxIGs may be a practical data structure for some technologies where MUX gates are of similar cost as AND and XOR gates, such as memristors [OKR14], quantum-dot cellular automata (QCA) [KA22], and pass transistor logic [SB00]. Although the MUX gate itself is functionally complete without inverters, we still use complemented edges to represent cost-free inverters in the network to be more memory-efficient. This can be disabled (i.e., $\neg x$ has to be implemented as $\text{MUX}(x, 0, 1)$) and the MUX-based resynthesis algorithm can also be adjusted accordingly, if desired. A MUX gate can implement the 2-input AND, OR, and XOR functions, thus MuxIGs are more compact than XAGs. Though conceptually similar, MuxIGs are different from BDDs [Ake78]. In BDDs, S-inputs can only be primary variables, whereas in MuxIGs, S-inputs can be connected to the output of any other MUX gates in the network. Thus, MuxIGs are more general than BDDs. In this section, we propose a MUX-based resynthesis algorithm that can be used to optimize MuxIGs.

Due to the natural characteristics of the MUX gate, our MUX-based resynthesis algorithm is designed with a combination of ideas from AND- and MAJ-based resynthesis. First, we observe that similar to resynthesizing with MAJ gates, we seek to select or construct functions resembling the target to be placed at the T- and E-inputs of a MUX gate, subject to a care function depending on the function at its S-input. Thus, we also normalize divisor functions and adopt the bit-counting-based ranking and selection of divisors as in MAJ-based resynthesis. Second, when there are some care bits not covered, unlike MAJ-based resynthesis, the expansions on the T- and E-inputs are independent of each other. For a MUX gate with care function c, once the S-input s is selected, the care function at the T-input is $c_t = c \wedge s$ and the care function at the E-input is $c_e = c \wedge \neg s$. Thus, we adopt the recursive decomposition similar to that in AND-based resynthesis to expand on T- or E-inputs until all care bits are covered. To avoid re-normalizing divisors and to simplify the computation, we do not expand on the S-input once it is selected.

Algorithm 4.3 illustrates the MUX-based resynthesis algorithm. First, the set N of normalized

Algorithm 4.3: Heuristic MUX-based resynthesis algorithm.

```
Input: target function $f$, care function $c$, divisor functions $G = \{g_1, \dots, g_n\}$
Output: dependency circuit $H$
1  $N \leftarrow normalize(G, f)$
2  return $resynthesize(c)$
3
4  Function $resynthesize($care $c)$:
5      $t \leftarrow \mathrm{argmax}_{l \in N} \textsc{Ones}(l \wedge c)$
6      if $\textsc{Ones}(\neg t \wedge c) = 0$ then
7          return $t$
8      $S \leftarrow \mathrm{argmin}_{l \in \{g, \neg g : g \in G\}} \textsc{Ones}(\neg t \wedge l \wedge c)$
9      $s \leftarrow \mathrm{argmin}_{l \in S} \textsc{Ones}(\neg l \wedge c)$
10     if $\textsc{Ones}(\neg s \wedge c) = 0$ then
11         $e \leftarrow 0$
12     else
13         $e \leftarrow \mathrm{argmax}_{l \in N} \textsc{Ones}(l \wedge \neg s \wedge c)$
14         if $\textsc{Ones}(\neg e \wedge \neg s \wedge c) > 0$ then
15             $e \leftarrow resynthesize(\neg s \wedge c)$
16     if $\textsc{Ones}(\neg t \wedge s \wedge c) > 0$ then
17         $t \leftarrow resynthesize(s \wedge c)$
18     return MUX$(s, t, e)$
```

divisors is derived using Equation (4.17) (line 1). The unchanged set N is then available and used throughout the algorithm along with the original set of divisors G. The recursive algorithm starts with the given top-level care function c (line 2). In line 5, a literal t covering the most care bits is chosen from N as the T-input. If all care bits are covered by t, then it is a 0-resyn and is returned (lines 5-6). Otherwise, we continue to choose a literal s from G as the S-input using two criteria: Literals S whose (cared) 1-bits overlap the least with the 0-bits of t are prioritized (line 8). If there are more than one literal in S, then the literal with the least 0 in the cares bits is chosen (line 9). The first criterion aims at reducing uncovered care bits at the T-input, whereas the second criterion aims at reducing the care bits to be covered at the E-input. If the selected s has no cared 0-bit, then the function at the E-input does not matter and we choose constant 0 as the E-input, assuming it has the lowest cost (lines 10-11). Otherwise, similar to choosing t, a literal e covering the most care bits is chosen as the E-input (line 13). Although the philosophy behind the choice of t and the choice of e is the same, there is a difference in their evaluations: When choosing t, the S-input is not selected yet, thus only the care function c for the gate is considered. However, when choosing e, the S-input s is already decided, thus the more precise care function at the E-input $c_e = c \wedge \neg s$ is considered. Finally, we check if the care bits at the T- and E-inputs are all covered by t and e, respectively, and recursively expand on the inputs using their care functions if not so (lines 16-17 and 14-15, respectively).

Table 4.1: Comparison of AIG resynthesis algorithms.

	1) $k = 4$, $\max m \geq 1$		2) $k = 6$, all problems		3) $k = 6$, $\max m \geq 4$	
#Probs	128312		337155		22691	
Avg. n	6.55		14.16		7.18	
Avg. $\max m$	1.53		0.70		4.18	
	SAT	Ours	Enum.	Ours	SAT	Ours
#Sols	920	990	1248	1589	522	465
Avg. m	1.72	1.71	1.97	2.61	4.17	4.38
Avg. overhead	–	0.00	–	0.05	–	0.16
	–	(0%)	–	(1%)	–	(3%)
Tot. time (s)	43.12	0.11	0.28	0.34	638.21	0.10

4.8 Experimental Results

In this section, we test the performance and efficiency of the proposed resynthesis algorithms on sets of real resynthesis problems extracted from the EPFL benchmarks [AGD15] by resubstitution (Section 4.8.1). We also demonstrate in Section 4.8.2 the effectiveness of using resynthesis as the core of an high-effort optimization to further optimize highly-optimized benchmarks. The experiments were conducted on a laptop with Apple M1 Pro chip and 32 GB RAM.

4.8.1 Extracted Resynthesis Problems

As the core of peephole optimization, it is more meaningful to compare different resynthesis approaches using real resynthesis problems in their general form, with arbitrary divisor functions coming into play. In this section, we test our heuristic resynthesis algorithm on sets of resynthesis problems extracted from the EPFL benchmark suite. The benchmarks are preprocessed by running the script `compress2rs` in ABC [BM10] once to rule out most optimizations that are easier to identify. To extract resynthesis problems, for each node (root) in the benchmarks, a reconvergence-driven cut [MB06] of size $k = 4$ or 6 is computed and used as the basis to obtain local functions of nodes supported by the cut. The function of the root node is the target of the resynthesis problem and the functions of all nodes supported by the cut, including the cut leaves, are divisors. The care set is derived by computing (local) satisfiability don't cares from a larger cut of size 12. A size limit $\max m$ is given along with the resynthesis problem, determined by the root's MFFC size minus 1.

Three sets of AIG resynthesis problems are considered in Table 4.1:

1. First big column: A subset of problems extracted using cut size $k = 4$ (thus truth table length $l = 2^k = 16$) where the size limit is at least 1.
2. Second big column: A subset of problems extracted using cut size $k = 6$ (thus truth table

length $l = 2^k = 64$) where the size limit is at most 3.

3. Third big column: A subset of problems extracted using cut size $k = 6$ where the size limit is at least 4.

The total number of resynthesis problems ("#Probs"), the average number of divisors per problem ("Avg. n"), and the average size limit ("Avg. max m") are listed for each set in the upper half of Table 4.1. We compare our AND-based heuristic resynthesis ("Ours") against SAT-based exact synthesis [Haa+20] ("SAT", Section 4.4.2, conflict limit = 10000) and enumeration-based method [MB06] ("Enum.", Section 4.4.3, up to 3 gates). The number of solutions found within the size limit ("#Sols"), the average number of gates in the dependency circuits found ("Avg. m"), the average overhead comparing to the optima ("Avg. overhead"), and the total runtime in seconds ("Tot. time") are listed for each method.

We observe from this experiment that resynthesis problems requiring larger dependency circuits do exist in real benchmarks. Both SAT and enumeration are exact algorithms, meaning that the solutions they give, if any, are always optimal. However, the optimality of SAT-based exact synthesis comes with the cost of a much higher runtime, and enumeration, although being fast, can only solve problems with small solutions. In 2), the 341 more problems solved by our heuristic than enumeration are cases where a solution cannot be found within three gates and the recursive decomposition described in Section 4.5.3 is necessary. The quality degradation of our heuristic is zero for smaller dependency circuits ($m \leq 3$), and is still very small (3%) for medium-sized dependency circuits for which SAT-based synthesis needs a long time to find the optimal solution.

4.8.2 Resynthesis as the Core of High-Effort Optimization

To demonstrate the practical application of the proposed heuristic resynthesis algorithms in high-effort optimization, we use them as the core component in the simulation-guided resubstitution framework [Lee+22] and perform experiments on benchmarks that are already optimized by state-of-the-art size optimization flows. The resubstitution framework computes, for each target node as the root, a reconvergence-driven cut of at most 8 nodes to collect up to 150 divisors supported by the cut. Functions of the target and divisor nodes are estimated by global simulation using about 1000 simulation patterns.

AIG

For AIG size optimization, the script `compress2rs` in ABC [BM10] is considered as the state-of-the-art flow, which comprises 18 commands including balancing, resubstitution, rewriting, and refactoring with different hyper-parameters. In Table 4.2, after listing the benchmark names and their original size, the size reduction in terms of percentage number of gates ("Red.") and runtime ("Time") of four optimization settings are presented: Column "O → A" applies `compress2rs` once on the original benchmarks; we call the resulting set of opti-

Table 4.2: AND-based heuristic resynthesis as the core of simulation-guided resubstitution applied on highly-optimized benchmarks.

AIG: O = Original, A = `compress2rs` ×1, B = `compress2rs` ×∞									
Benchmark		O → A		A → Ours		A → B	B → Ours	
Name	Size (#gates)	Red. (%)	Time (s)	Red. (%)	Time (s)	Red. (%)	Red. (%)	Time (s)
adder	1020	12.55	0.08	0.00	0.00	0.00	0.00	0.00
bar	3336	5.85	0.27	2.58	0.04	0.00	2.58	0.04
div	57247	63.80	3.64	0.89	0.23	1.05	0.00	0.63
hyp	214335	4.57	30.03	0.15	14.93	0.16	0.05	14.44
log2	32060	8.95	5.08	1.73	5.41	0.56	1.54	6.10
max	2865	1.15	0.18	0.00	0.01	0.28	0.00	0.02
multiplier	27062	10.07	3.53	0.10	0.28	0.09	0.01	0.27
sin	5416	7.33	0.97	1.35	0.44	1.00	1.19	0.58
sqrt	24618	25.85	2.87	0.30	4.39	0.01	0.26	4.39
square	18484	14.03	2.61	0.66	0.13	0.57	0.09	0.06
arbiter	11839	0.00	1.42	0.00	0.15	0.00	0.00	0.28
cavlc	693	8.37	0.19	4.25	0.09	2.20	3.06	0.16
ctrl	174	48.28	0.04	0.00	0.00	0.00	0.00	0.00
dec	304	0.00	0.06	0.00	0.00	0.00	0.00	0.00
i2c	1342	20.34	0.12	2.90	0.02	5.05	2.17	0.02
int2float	260	19.62	0.05	0.96	0.03	0.96	0.48	0.06
mem_ctrl	46836	6.22	5.21	15.95	1.76	12.13	14.09	2.03
priority	978	52.35	0.07	0.64	0.00	8.15	0.23	0.01
router	257	28.79	0.04	20.77	0.00	20.77	9.66	0.00
voter	13758	42.24	1.58	0.18	0.02	0.13	0.08	0.05
Average		19.02	2.90	2.67	1.40	2.66	1.77	1.46
Total gain		71402		8460		6360	6257	

mized benchmarks A. Column "A → Ours" applies simulation-guided resubstitution using our heuristic AND-based resynthesis on the benchmark set A. Column "A → B" applies more times of `compress2rs` on A until no more size reduction is observed for at least 5 consecutive times; we call this set of benchmarks B. Column "B → Ours" applies our resubstitution on the benchmark set B. In the last row, "Total gain" lists the total number of reduced gates, summed over all benchmarks.

Comparing "A → Ours" and "A → B", we can observe that, on top of the benchmark set A that is already optimized, our high-effort optimization achieves similar "leftover" size reduction as the best `compress2rs` can do. Moreover, column "B → Ours" shows that our approach can still squeeze 1.78% more size reduction out of the highly-optimized benchmark set B. In both "A → Ours" and "B → Ours", the runtime of our high-effort optimization is comparable with `compress2rs`.

Experiments on XAG, MIG and MuxIG optimization all use the optimized benchmark set A as the starting point (column "AIG" in Tables 4.3 and 4.4). Besides size reduction percentage ("Red.") and total runtime ("Time"; for Columns XAG and MIG, time for `compress2rs` is excluded), the runtime spent by our heuristic algorithms in solving the resynthesis problems is also listed ("T_{resyn}").

XAG

For XAG optimization, we first apply the LUT mapping command `&if` in ABC with K (number of inputs per LUT) set to 2, followed by the interpolation-based LUT resubstitution command `&mfs` [Mis+11b] to obtain XAG benchmarks (column "XAG" in Table 4.3; note that a 2-LUT network is essentially an XAG). Then, in column "XAG → Ours" we apply simulation-guided resubstitution using our AND-based resynthesis with XOR enabled, and 2.86% size reduction is obtained from the set of optimized XAGs within similar runtime as optimizing and transforming into XAGs.

MIG

As the state-of-the-art MIG optimization flow, we apply three times graph (re-)mapping [Tem+22] from the optimized AIGs, followed by enumeration-based MIG resubstitution [Rie+18] repeated until no more size reduction is observed (column "MIG" in Table 4.4). Then, similarly, simulation-guided resubstitution using our MAJ-based resynthesis is applied, which obtains 2.45% size reduction on top of highly-optimized benchmarks within a faster runtime (column "MIG → Ours" in Table 4.4).

Table 4.3: AND-XOR-based heuristic resynthesis as the core of simulation-guided resubstitution applied on highly-optimized benchmarks.

AIG = `compress2rs`, XAG = `compress2rs; &if -K 2; &mfs`						
	AIG	XAG		XAG → Ours		
Benchmark	Size (#gates)	Size (#gates)	Time (s)	Red. (%)	Time (s)	T_{resyn} (s)
adder	892	637	0.04	0.00	0.00	0.00
bar	3141	3141	1.16	2.10	0.04	0.03
div	20725	16791	0.13	0.40	0.63	0.06
hyp	204533	160201	72.60	5.03	47.55	0.46
log2	29192	23966	19.58	1.55	2.15	0.22
max	2832	2832	0.12	0.00	0.02	0.01
multiplier	24337	18571	10.59	0.12	0.23	0.13
sin	5019	4263	11.37	2.18	0.54	0.04
sqrt	18255	14381	0.13	12.79	3.41	0.05
square	15891	12450	9.80	0.10	0.07	0.04
arbiter	11839	11839	29.94	0.00	0.34	0.13
cavlc	635	634	0.12	5.21	0.23	0.22
ctrl	90	90	0.01	4.44	0.00	0.00
dec	304	304	0.01	0.00	0.00	0.00
i2c	1069	1062	0.08	3.48	0.03	0.02
int2float	209	208	0.02	2.88	0.05	0.04
mem_ctrl	43924	38241	61.50	10.11	2.28	1.04
priority	466	443	0.07	1.13	0.02	0.01
router	183	143	0.01	5.59	0.01	0.00
voter	7946	5717	4.23	0.12	0.53	0.02
Average			11.08	2.86	2.91	0.13

MuxIG

Finally, as there is not yet much research on MuxIG, we transform the optimized AIGs directly into MuxIGs by replacing AND gates with MUX gates with a constant input. Then, in column "MuxIG, ours" in Table 4.4, simulation-guided resubstitution using our MUX-based resynthesis successfully reduces the sizes of these MuxIGs by 20.24% by identifying MUX functions in the networks. It is worth noting that although the runtime for the largest benchmark *hyp* seems to be long, the time spent in the resynthesis algorithm takes only 1% and most of the time is spent in proving the validity of the identified optimization choices.

4.9 Summary

In this chapter, three heuristic resynthesis algorithms are proposed, targeting networks based on AND, MAJ, and MUX gates. The common characteristic of the proposed algorithms is that they are efficient heuristics without superlinear scalability concerns. Table 4.5 compares the proposed heuristics with other existing methods. All methods compared solve the resynthesis

Table 4.4: MAJ-based and MUX-based heuristic resynthesis as the core of simulation-guided resubstitution applied on highly-optimized benchmarks.

AIG = `compress2rs`, MIG = `compress2rs` + map ×3 + resub ×∞									
	AIG	MIG		MIG → Ours			MuxIG, ours		
Benchmark	Size (#gates)	Size (#gates)	Time (s)	Red. (%)	Time (s)	T_{resyn} (s)	Red. (%)	Time (s)	T_{resyn} (s)
adder	892	384	0.11	0.00	0.00	0.00	28.48	0.03	0.01
bar	3141	2594	0.29	0.23	0.03	0.03	43.36	0.07	0.02
div	20725	12565	0.93	0.26	0.32	0.11	39.24	2.64	0.10
hyp	204533	127877	13.01	2.89	9.10	0.86	21.56	104.69	1.05
log2	29192	23643	3.00	2.26	6.41	0.34	14.92	21.23	0.24
max	2832	2210	0.32	0.00	0.03	0.03	28.32	0.08	0.02
multiplier	24337	18700	1.76	1.39	0.34	0.20	19.13	4.51	0.20
sin	5019	4018	0.81	1.27	0.19	0.07	15.06	0.77	0.05
sqrt	18255	12513	1.09	0.72	3.25	0.16	20.36	4.35	0.11
square	15891	9573	1.03	0.78	0.08	0.05	30.87	1.06	0.08
arbiter	11839	6866	1.38	2.14	0.17	0.14	1.08	0.42	0.33
cavlc	635	541	0.83	1.48	0.02	0.02	14.02	0.02	0.01
ctrl	90	80	0.21	1.25	0.01	0.01	15.56	0.00	0.00
dec	304	304	0.09	0.00	0.01	0.01	0.00	0.01	0.01
i2c	1069	951	0.12	2.00	0.02	0.02	19.36	0.02	0.01
int2float	209	190	0.09	4.74	0.01	0.01	12.44	0.00	0.00
mem_ctrl	43924	38179	3.86	8.91	2.24	1.24	23.23	2.78	0.97
priority	466	449	0.10	4.01	0.01	0.01	13.30	0.01	0.00
router	183	170	0.07	11.18	0.00	0.00	21.86	0.00	0.00
voter	7946	4729	0.53	3.55	0.05	0.03	22.73	0.29	0.04
Average			4.38	2.45	1.11	0.17	20.24	7.15	0.16

problem with incompletely-specified functions (Problem Formulation 3), except for looking up in an optimal database, which only solves a subset of resynthesis problems where divisors are projection functions and all functions are completely-specified. All algorithms are sound, but only database look-up, SAT-based exact synthesis, and enumeration are complete and guarantee optimality. As a compromise, these exact methods have a rather high complexity (except for database) and are practically limited by the number of divisors (n), the size of dependency circuit (m), and/or the truth table length (l). In contrast, although the proposed heuristics do not guarantee optimality, their complexities are linear in all variables (or only quadratic in n for AND-based resynthesis) and are thus practically unlimited.

Experimental results show that the proposed heuristic resynthesis serve as an important component in high-effort peephole optimization, achieving, on average, about 2-3% more size reduction on benchmarks that are already highly-optimized, within manageable runtime. The key to finding these hidden optimization opportunities is the heuristics' capability to solve resynthesis problems with more divisors (scalability in n), having larger solutions (scalability in m), and where functions are given as longer simulation signatures (scalability in l).

Table 4.5: Comparisons of existing and proposed resynthesis algorithms.

	Database	SAT-based (SSV encoding)	Enumeration	Akers'	Proposed heuristics
	[MCB06]	[Haa+20; RMS20]	[MB06; Ama+18; Rie+18]	[Ake62]	
Support of divisors	no	yes	yes	yes	yes
Support of incomplete functions	no	yes	yes	yes	yes
Soundness	yes	yes	yes	yes	yes
Completeness	yes	yes	yes	no	no
Optimality	yes	yes (if solved iteratively)	yes	no	no
Complexity	$\mathcal{O}(1)$	#vars: $\mathcal{O}(m((n+m)^\kappa + l))$ #clauses: $\mathcal{O}(m(n+m)^\kappa)$	$\mathcal{O}(n^{(\kappa-1)m+1} l)$	$\mathcal{O}(n^2 m l^2)$	AND-based: $\mathcal{O}(n^2 m l)$ MAJ- and MUX-based: $\mathcal{O}(nml)$
Practical limits	$n = k \leq 4$	$n + m \leq 10, k \leq 6$	$m \leq 3$	unknown	no limit

n: number of divisors, m: number of gates in dependency circuit, k: number of variables of target and divisor functions, $l = 2^k$: length of truth tables, κ: number of fanins per gate ($\kappa = 2$ for AIG and XAG; $\kappa = 3$ for MIG and MuxIG)

5 Design Space Exploration

5.1 Motivation

The general case of the Boolean optimization problem is intractable, such that academic as well as industrial tools rely on well-tuned heuristics. Boolean optimization algorithms such as rewriting, factoring, and resubstitution [Rie+19a; Lee+22] have been revisited several times and have been improved in scalability and achievable optimization quality. Combining the individual algorithms into an efficient Boolean optimization flow, however, is rarely addressed and requires careful parameter tuning.

As a remedy, recent research proposals suggest data-driven *Artificial Intelligence* (AI) to guide logic synthesis flows and improve overall QoR. An intelligent agent powered by AI could be capable of smarter decision-making by controlling when to run and stop logic optimization while considering trade-offs and conflicting QoR goals [Net+22; Per+21]. Modern AI technology, however, has its own challenges: computational demands are often extraordinarily high, large amounts of training data are required, aggressive learning policies may result in biased and unexplainable decision-making, sophisticated training, and learning approaches require AI experts to design, tune, and maintain.

Moreover, with the development of beyond-CMOS emerging technologies, unconventional circuit properties, design constraints, and cost functions need to be considered in design automation. For example, *Spin Torque Majority Gate* (STMG) [NBG11] circuits are based on majority gates and inverters are expensive, thus MIG [AGD16] instead of AIG is a better logic network abstraction. AQFP [Tak+13] is also based on majority gates, and it imposes additional path-balancing and fanout-branching constraints. *Field-coupled Nanocomputing* (FCN) [AB14] is a family of nanotechnologies whose physical design requires the circuit to be planarized, in addition to path-balancing and fanout-branching. Although these constraints may be dealt with after technology mapping, research has shown that tailored logic optimization algorithms considering specialized cost metrics early on yield better QoR. However, carefully-tuned optimization flows for individual technologies are even more rarely researched, as such work would require experts in both the technology and logic synthesis (and AI).

In this chapter, we propose a simpler design space exploration approach that takes a combinational gate-level circuit represented as input, evaluates its characteristics, and makes decisions about what optimizing transformations to apply as it proceeds. Our goal is to provide an easily adaptable solution, customizable for various applications, when the best-achievable QoR is of interest and higher runtime is acceptable. Restart and bailout strategies are used in the exploration procedure as a mechanism to retry if a logic minimum has been reached and to terminate optimization early if QoR deviates too much from a desired quality goal.

5.2 Related Works

Logic optimization flows are fixed sequences of optimizing transformations. While many research works focus on improving the performance and quality of individual transformations, complete optimization flows are rarely proposed. The problem of finding a sequence of optimizing transformations that achieves the best results for a given benchmark suite is only recently investigated using techniques from machine learning (ML) [Per+21; YXM18; Net+22] and Bayesian optimization [Gro+22]. These works arrange existing technology-independent optimizing transformations to reduce the area and delay of the final netlist as much as possible, where each optimizing transformation maintains a local view on the logic, e.g., in the form of sliding windows, and implements a well-known scalable logic optimization. Alternatively, algorithms based on global optimization principles such as simulated annealing [MJV00] and evolutionary algorithms [Fis+10; FDK11] achieve better logic compaction, but they are rarely considered in practice due to their massive computational demands.

5.3 Overview

An overview of the on-the-fly design space exploration algorithm is outlined in Algorithm 5.1. Like most logic network optimization algorithms, it takes an original network as input and outputs an optimized network. Additionally, there are three custom functions a user should specify: cost evaluation, decompressing, and compressing scripts, which will be further described in Sections 5.4 and 5.7.

There is an outer loop (lines 3-19) and an inner loop (lines 8-17) in Algorithm 5.1. In the following, we call an iteration of the outer loop a *restart* and an iteration of the inner loop a *step*. Furthermore, an execution of `decompress` (line 9) or `compress` (line 10) is called a *script*, which may contain one or more algorithms or transformations.

In each restart, the network is restored to the original one, and a new random engine seeded with a different seed is generated (line 6). The number of restarts is defined by the user (parameter *num_restarts*). The best network having the smallest cost in all restarts is recorded and eventually output by the algorithm (lines 18-20). Each restart has its own timer to upper-bound the runtime (line 7).

Algorithm 5.1: On-the-fly design space exploration

```
Input: Original network $N_0$
Output: Optimized network $N_{\text{best}}$
Custom functions: cost, decompress, compress
Parameters: $num\_restarts, max\_steps, max\_no\_impr, timeout, init\_seed$
1  $N_{best} \leftarrow N_0$.copy()
2  $R_1 \leftarrow$ random_engine($init\_seed$)
3  for $restart = 1$ upto $num\_restarts$ do
4      $N_{best\_inner} \leftarrow N_0$.copy()
5      $N_{curr} \leftarrow N_0$.copy()
6      $R_2 \leftarrow$ random_engine($R_1$.rand())
7      $elapsed\_time \leftarrow 0$; start_timer($elapsed\_time$)
8      for $step = 1$ upto $max\_steps$ do
9          decompress($N_{curr}$, $R_2$.rand())
10         compress($N_{curr}$, $R_2$.rand())
11         if cost($N_{curr}$) < cost($N_{best\_inner}$) then
12             $N_{best\_inner} \leftarrow N_{curr}$.copy()
13             $last\_impr \leftarrow step$
14         else if $step - last\_impr \geq max\_no\_impr$ then
15             break
16         else if $elapsed\_time \geq timeout$ then
17             break
18     if cost($N_{best\_inner}$) < cost($N_{best}$) then
19         $N_{best} \leftarrow N_{best\_inner}$.copy()
20 return $N_{best}$
```

Each step consists of a call to a decompressing script followed by a call to a compressing script, which are both randomized. After these transformations are done, the network cost is evaluated. The current network is recorded if its cost is the best seen among the steps executed so far in the current restart (lines 11-12). The inner loop breaks if there have been *max_no_impr* steps executed without seeing a better network (lines 14-15), or if the timeout limit has reached (lines 16-17).

In the remainder of this section, we explain why we believe such algorithmic design helps achieve better design space exploration.

5.4 Escaping Local Optimum

Although a user of our algorithm has the freedom to define any set of decompressing and compressing scripts, we encourage them to classify possible transformation algorithms into two categories and have good candidates in both. A decompressing script should be a script that dramatically restructures the network and likely increases its size and depth. A prominent example of a decompressing script is LUT mapping followed by naive resynthesis to convert back into the original representation (e.g. AIG or MIG). Another example, when the representation is an MIG, is randomly breaking each majority gate into four using the relation

$$\mathrm{MAJ}(a,b,c) = (a \wedge b) \vee (c \wedge (a \vee b)) = \mathrm{MAJ}(\mathrm{MAJ}(a,b,0),\mathrm{MAJ}(c,\mathrm{MAJ}(a,b,1),0),1). \quad (5.1)$$

The purpose of decompressing is to create the possibility of escaping from local optima. Imagine if the design space of all feasible networks is projected to the x axis and the y axis is the cost of each network. Such a curve is very likely not convex and many valleys of local minima exist. When we are stuck at a local minimum, decompressing scripts help us climb up the hills and potentially reach a better local minimum afterward.

In contrast, a compressing script is a sequence of algorithms that attempts to optimize for the given cost metric. Examples of compressing scripts include well-known logic optimization algorithms such as rewriting, balancing, refactoring, resubstitution, graph remapping, etc. The aim of compressing scripts is to converge to a local minimum. By interleaving decompressing and compressing scripts, our algorithm may explore different local optima in the design space, instead of being trapped in the nearest local optima when only applying one optimization algorithm.

5.5 Stretching Out in the Design Space

Consider the original network N_0 and a certain optimized network N_{best} to be reached, they may be far away in the design space, and a long sequence of transformations is required to get from N_0 to N_{best}. Thus, our design space exploration strategy aims at stretching far out and really performing long transformation sequences. The key to such an aim is that in the inner loop, even if the cost is getting much worse, there is no mechanism to backtrack to the previous best result or to retrieve the original network. A design space exploration strategy that tries many different combinations of transformation sequences but frequently backtracks would explore the design space more densely near the original network, but less likely to reach out to further points.

5.6 On-the-fly Exploration

Being able to try long sequences of transformations is not enough. The next big question is: What kind of sequence leads to better results? Although machine-learning-based research and human expert experiences give some insights, we argue that the answer is different for different benchmarks and different cost metrics. Instead of pre-defining particular sequences, our algorithm simply performs random walks. The purpose of the outer loop is to mitigate the possibility of a “bad” random seed leading to unsatisfactory results and to increase the chance of meeting at least one “good” random sequence in all restarts. We call such strategy an *on-the-fly* exploration because we do not know the best transformation sequence in advance, but discover it on the fly during exploration.

5.7 Customization

Aiming at applications to emerging technologies with diverse logic representations, dedicated algorithms, and cost evaluation metrics, our algorithm is customizable in these aspects.

- Logic representation: As long as the transformation scripts and cost evaluation function are compatible, there is no limitation on the data structure of N_0. Although this chapter focuses mainly on network optimization, it is also possible to use other logic representations such as two-level forms.
- Decompressing and compressing scripts: To set up the algorithm, the user must provide a nonempty set of decompressing scripts and a nonempty set of compressing scripts. When the functions `decompress` and `compress` (line 9 and 10 in Algorithm 5.1) are called, one of the scripts in the respective set is randomly chosen. The user may also define the probability of each script being chosen, preferring some scripts over the others. Moreover, how randomness is involved in the scripts is also customizable. For example, a user may define that the cut size to be used in resubstitution is randomly chosen within a range.
- Cost evaluation: Most importantly, the cost evaluation function is customized. Such a function should take a network as input and output a number. It should not modify the network, but it may execute complicated algorithms to compute the cost.

Besides the custom functions, there are also some parameters users may set according to their needs.

- *num_restarts*: This parameter defines how many different transformation sequences, or exploration paths, will be tried randomly. We will experiment on the impact of this parameter in Section 5.8.3.
- *max_steps, max_no_impr, timeout*: These parameters define the optimization effort of each restart. Particularly, *max_no_impr* defines how many steps without seeing any improvement in the cost the algorithm will tolerate before bailing out from the current exploration path, and *timeout* defines the runtime budget.
- *init_seed* is the user-specified initial random seed used to generate different seeds to be used in each restart. This parameter is only used to ensure deterministic and reproducible results of the algorithm. When *num_restarts* is sufficiently large (Section 5.8.3), different *init_seed* should give similar results, and tuning of this parameter should not be needed.

Table 5.1: Comparison of MIG size against previous works.

	Map [Tem+22]	Flow [TD24]	DSE [Ours]		
Bench.	Size	Size	Size	Impr.	Depth
adder	384	384	384	-	129
bar	2588	2433	1906	21.7%	15
div	36858	12462	12368	0.8%	2251
hyp	137048	115541	115539	0.002%	9129
log2	24295	22010	22008	0.01%	184
max	2171	2190	1939	11.5%	172
multiplier	19299	17112	17112	-	137
sin	4196	3870	3869	0.03%	124
sqrt	17355	12357	12247	0.9%	2156
square	11924	8138	8089	0.6%	126
Total (arith.)	256118	196497	195461	0.53%	14423
arbiter	-	6711	792	**88.2%**	108
cavlc	-	492	374	24.0%	16
ctrl	-	74	60	18.9%	8
dec	-	304	304	-	3
i2c	-	871	636	27.0%	16
int2float	-	172	115	33.1%	9
mem_ctrl	-	32097	6886	**78.5%**	26
priority	-	406	337	17.0%	23
router	-	147	97	34.0%	13
voter	-	4555	3894	14.5%	32
Total (all)	-	242326	208956	13.8%	14677

5.8 Experimental Results

In this section, we present experimental results on the problem of MIG size optimization as an example. The EPFL benchmark suite [AGD15] is used.

5.8.1 Application to MIG Optimization

Table 5.1 compares a state-of-the-art MIG restructuring algorithm, graph remapping [Tem+22] (Map), the current best MIG size results seen in the literature produced by an optimization flow [TD24] (Flow), and the new best results achieved by our design space exploration (DSE). The MIG sizes (number of gates) are listed for all of the three as the main comparison, and the MIG depth is additionally listed in DSE for reference. Column "Impr." computes the improvement percentages of MIG size comparing our DSE to SoTA Flow. The benchmark suite is divided into arithmetic circuits (upper half) and control circuits (lower half), and the sum of arithmetic benchmarks as well as all benchmarks are listed separately. Data of the control circuits for Map were omitted in the table because they were not presented in [Tem+22].

From Table 5.1, we observe the improvements made by extending from a single algorithm to a fixed flow, and finally to an exploration of a portfolio of different flows. Overall, our new best result improves over state of the art by 13.8%.

Another application of the proposed design space exploration on AQFP optimization will be presented in Chapter 10.

5.8.2 Design Space Exploration

We take the benchmark "arbiter" from the MIG optimization experiment and plot the processes of three restarts in Figure 5.1 as an example illustration of design space exploration. The optimization goal is set to minimize MIG size (y-axis), and the MIG depth is used as the x-axis of the plot to help distinguish different networks seen in the process. Both axes are plotted on a logarithmic scale. Only the networks causing an update to $N_{\text{best\_inner}}$ are recorded. From this figure, we can observe the different paths taken by the design space exploration algorithm. The third restart (green) ends up with the best $N_{\text{best\_inner}}$ and is taken as the final N_{best}.

5.8.3 Importance of Random Restarts

To investigate the influence of different random seeds used in each restart, we plot the best-seen network in 50 restarts in the same run. The benchmark "priority" from the EPFL benchmark suite is used and optimized for MIG size. In Figure 5.2, the y-axis is MIG size (optimization goal) and the x-axis is MIG depth (a second network trait). Both axes are plotted on a logarithmic scale. Each blue cross is a local optimum $N_{\text{best\_inner}}$ recorded after 50 steps of transformation without improvement or when the inner loop times out, and the green square marks the best among the 50 restarts. The red circle is the initial network N_0, and the brown crosses are the results of fixed, predefined flows designed by human experts.

We observe from this experiment that there really exist many different local minima in the design space. Some of them are worse in both metrics, and some of them form a portion of the Pareto curve. As the algorithm is a random process, the order of encountering them is random. If *num_restarts* was set smaller, the chance of getting the same best local optimum is reduced.

However, there are not infinite local minima, and increasing *num_restarts* indefinitely may not always help find a better optimum. We have observed that for some benchmarks and settings, many restarts fall into the same few local minima.

5.9 Summary

This work presents an on-the-fly design space exploration algorithm that emphasizes long transformation sequences and restarts with different random decisions. The implementation is customizable for unconventional cost functions often seen in emerging technologies, as well as dedicated, customized optimization scripts. With the proposed design space exploration, we are able to improve over state-of-the-art QoRs on MIG and AQFP optimization problems.

We study the different trajectories of design space exploration and experimentally show that there may be many different local optima reachable by different flows found by the design space exploration algorithm. We argue that there does not exist a fixed universally-good flow that works well for all benchmarks so that the search for the best flow shall be done on the fly. As future work, we would like to experimentally demonstrate this claim by applying the best flow found for one benchmark on another benchmark.

Randomized decision is key to the proposed algorithm because it is the premise of forming different flows and taking different trajectories leading to different local optima. The algorithm would not work if there is only one unrandomized script provided. However, it remains an open question how many different scripts do we need. We conjecture that the more randomization involved, the wider the distribution of local optima we will get in a plot similar to Figure 5.2. In other words, better optima would become reachable, but there will be more worse optima as well. Further experiments are required to answer this question.

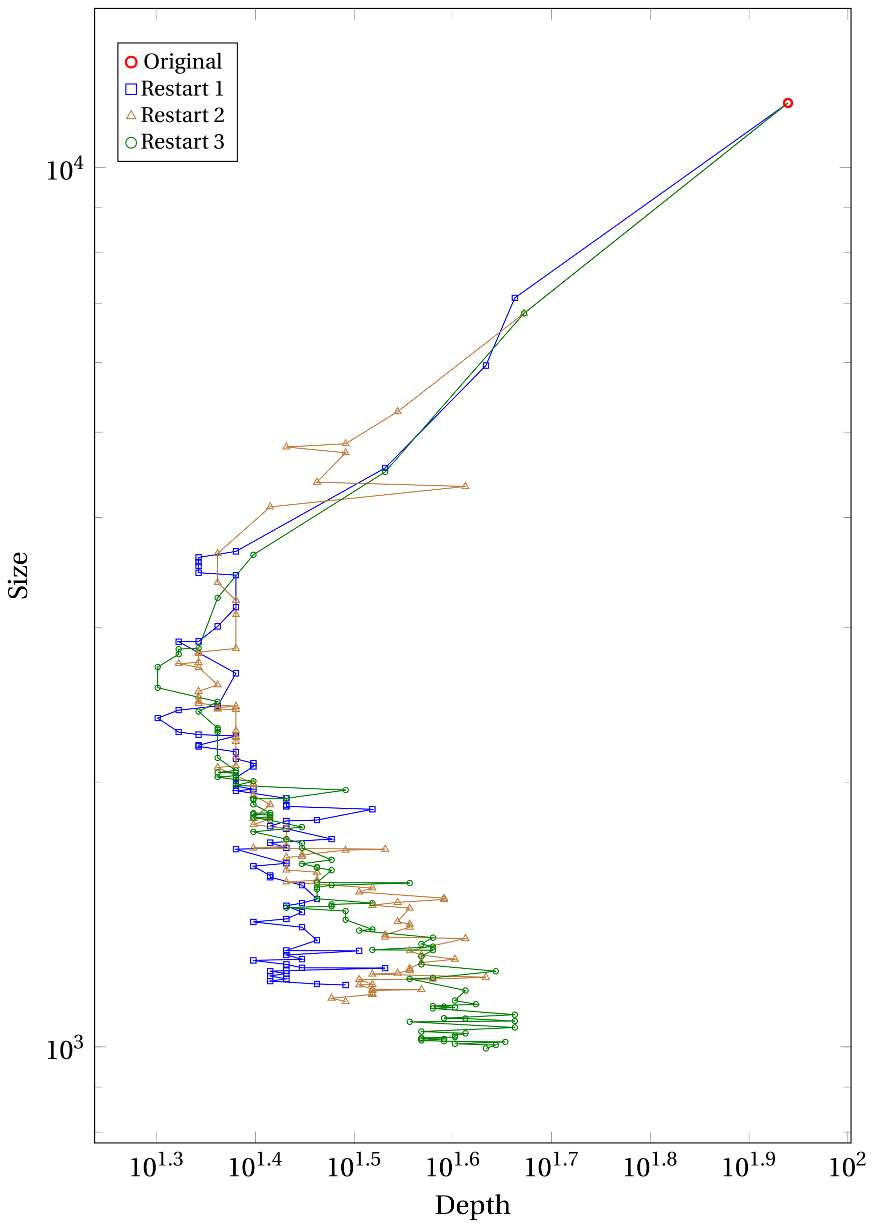


Figure 5.1: Three different paths in the design space taken by three restarts.

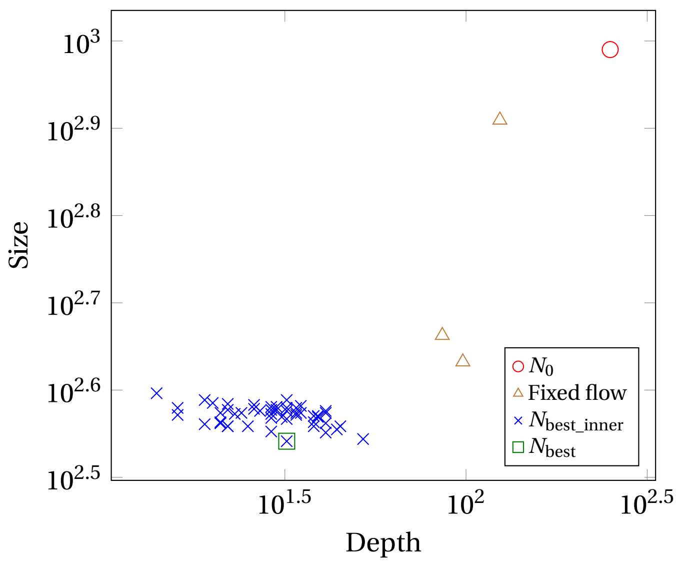


Figure 5.2: Local optima found by 50 restarts (×) compared to a fixed flow (△).

6 Testing and Debugging Logic Synthesis Algorithms

6.1 Motivation

The inherent complexity of these engines, optimized for many corner cases, makes logic synthesis algorithms susceptible to design and implementation errors. Moreover, algorithms are often only tested on fixed benchmark suites, such as the EPFL logic synthesis benchmarks [AGD15]. Due to numerous possibilities to implement the same Boolean function with different circuit structures, it is not rare that subtle faults slip through the development process and only show themselves when the algorithm is used in practice.

Motivated by the success of automated testing methods, we argue that directed testing approaches and bug-pointing tools specialized for logic synthesis applications can support the developers in detecting bugs earlier, can make implementations more robust, and ultimately lead to a reduction in the time and effort spent for debugging. Due to the large state space and homogeneity of the commonly used netlist formats, general-purpose testing and debugging tools often are incapable of providing the necessary performance to efficiently test implementations of logic synthesis algorithms. The C++ logic network library *mockturtle* [Rie+19b; Soe+22] has deployed a framework for unit testing, continuous integration on various operating systems and compilers, and a static code analysis engine controlled by user-defined queries to aid developers.

6.2 Scope

This chapter focuses on testing and debugging software applications, referred to as the *application under tests* (AUTs), that take a logic network, called a *testcase*, as an input. Prominent examples of such applications include implementations of logic synthesis algorithms such as rewriting [MCB06], resubstitution [Mis+11b], and technology mapping [Tem+22]. Methods to verify the correctness of the results, referred to as the *verification*, are assumed to be provided. They may come from several sources:

- Assertions within the program.
- Memory protection processes in the operating system checking for illegal memory access (typically raising segmentation faults).
- CEC [Mis+06a] of the output network against the input testcase (for logic optimization algorithms).
- Additional code checking coherence of the program's internal data structures, such as checking if the network is acyclic and checking the correctness of reference counts, etc.
- Another algorithm of the same purpose used to provide reference solutions (for problems having a unique correct solution).

A failing verification, e.g., a non-equivalent CEC result, indicates that a *defect* of the AUT is observed and the testcase used is said to be *failure-inducing*. The AUT combined with its verification is referred to as an *oracle*, and running the oracle with a testcase is an *oracle call*.

6.3 Related Works

6.3.1 Fuzz Testing

Fuzz testing [MFS90] is a software testing technique heavily used to detect security-related vulnerabilities and reliability issues. It is conceptually simple, yet empirically powerful. A fuzzing algorithm involves repeatedly generating testcases and using them to test the AUT. The idea of fuzz testing first appeared in 1990, when spurious characters in the command line caused by a noisy dial-up connection to a workstation led to, surprisingly, crashes of the operating system [MFS90]. Nowadays, the generation of testcases in fuzz testing algorithms often involves randomness, and the testcases are supposed to be beyond the expectation of the AUT.

Various taxonomies of fuzz testing algorithms have been developed. For example, black-box fuzzers [Lee+17] treat the AUT as a black-box oracle and only observe its input/output behavior, whereas white-box fuzzers [GLM08; CDE08] analyze some internal information of the AUT and generate testcases accordingly. Depending on the targeted types of AUTs, some fuzzers generate testcases based on predefined models or grammars [DRH14], whereas some other fuzzers mutate an initial *seed* testcase to generate more testcases [CWB15]. There are often some parameters to be set for the testcase generators. A series of fuzz-tests using testcases generated with a specific parameter configuration is called a *fuzz testing campaign* [Man+21].

6.3.2 Delta Debugging and Testcase Minimization

Given two versions of the code of a program, where the first version works but the second fails, *delta debugging* [Zel99] is a method originally proposed to extract a minimal set of changes (differences in the two versions of code) that causes the failure. The algorithm was later extended for minimizing failure-inducing testcases [ZH02].

The basic idea of delta debugging is binary searching and dividing the set of components, may it be the *delta* between two versions of code or the input testcase to a program, testing the program with the reduced set, keeping the subsets that preserve the failure, and increasing the granularity of division. The delta debugging algorithm (*ddmin*) guarantees to find a 1-minimal subset and requires, in the worst case, $n^2 + 3n$ oracle calls, where n is the size of the given set [ZH02].

Besides delta debugging being a generic method for testcase minimization, researchers have claimed that domain-specific testcase minimization techniques are more effective and efficient for some applications such as tree-structured inputs [MS06], compilers [Reg+12] and SMT solvers [KNP21]. Various open-source implementations of testcase minimization tools exist, including the general-purposed `delta`[1], `aigdd`[2] for the AIGER format, `ddSMT`[3] for the SMT-LIB v2 format, and the LLVM `bugpoint` tool[4]. Inspired by delta debugging, in this chapter, we aim to provide such an effective testcase minimization tool specialized for logic networks but not limited to AIGs.

6.4 Testing and Debugging Toolkit for Logic Synthesis Applications

6.4.1 Testcase Generation

We develop a fuzz testing framework for testing any application that takes a logic network as input. The AUT and the verification checks are provided as a combined oracle call, thus categorizing it as a black-box fuzzer. Although in some cases of fuzzing, testing with malformed testcases is key to testing the robustness of the AUT, this is not the case for our usage. In logic synthesis applications, detecting and rejecting malformed inputs, e.g. a cyclic network, are usually dealt with by the parsers instead of the logic synthesis algorithms. Nevertheless, as logic synthesis applications are often only tested with some common benchmark suites, our fuzzing framework still tests them with a larger input space beyond what they are usually tested with.

To generate random testcases, we propose three parameterized methods. These methods apply to any type of network having a finite set of possible gate types.

Random: Randomly generate nodes in topological order. This method is parameterized by the starting number of PIs x_0, the starting number of gates y_0, the number of networks z of the same configuration to generate, the increment of the number of PIs Δx, and the increment of

[1] https://github.com/dsw/delta
[2] https://github.com/arminbiere/aiger
[3] http://fmv.jku.at/ddsmt/
[4] https://llvm.org/docs/Bugpoint.html

the number of gates Δy. The generator starts by generating networks of $x = x_0$ PIs and $y = y_0$ gates and keeps a counter of how many networks have been generated. After generating z networks, the values of x and y are increased by Δx and Δy, respectively. Given the current values of x and y, a network is generated by:

1. Create x PIs.
2. Randomly decide on a gate type. Assume that the type requires κ fanins.
3. Randomly sample κ nodes (PIs or gates) that have been created.
4. Randomly decide for each fanin if it is complemented.
5. Create the gate. Repeat from step 2 if the number of gates is smaller than y.
6. Assign all nodes without fanout to be POs.

For network types with trivial-case simplifications (e.g., in AIGs, attempting to create an AND gate with identical fanins results in returning the fanin without creating a gate) and structural hashing enabled, the number of gates may not increase after step 4. Thus, the loop of steps 2 to 4 may iterate more than y times and the terminating condition is when the actual number of gates is y. If the parameters are set improperly, e.g., if $x = 1$, this might lead to an infinite loop.

Topology: Exhaustively enumerate all small-sized DAG topologies and randomly concretize them. This method is parameterized by the starting number of gates y_0, the lower r_l and upper r_h bounds on the PI-to-input ratio, and the number of networks z of the same configuration to generate. Upon initialization, the generator enumerates all isomorphic DAG topologies of $y = y_0$ vertices using an algorithm implemented in [Haa+20] and randomly shuffles them. Then, it starts by generating networks of the first topology and keeps a counter of how many networks have been generated. After generating z networks, the generator moves on to generating the next topology. After all topologies have been used to generate z networks, the value of y is incremented by 1 and topologies of the increased size are enumerated. Given a topology, which is specified by a DAG T with hanging inputs (i.e., the topology specifies how gates are connected to each other, but not how they are connected to PIs), a random network is concretized by:

1. Let i be the number of hanging inputs in T. Randomly decide on an integer x such that $r_l \cdot i \leq x \leq r_h \cdot i$. Create x PIs.
2. For each input of T, randomly decide on a PI to connect to.
3. For each vertex in T, randomly decide on a gate type.
4. For each edge in T, randomly decide whether it is complemented.

5. Assign the last gate to be a PO.

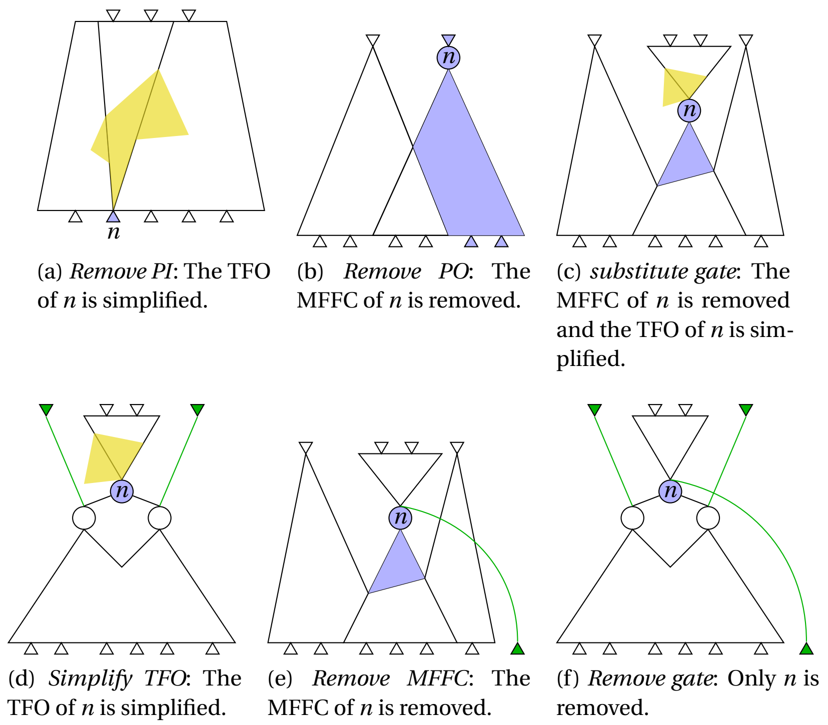
In step 1, lower values of x/i lead to a higher probability that the generated network reconverges on PIs, whereas higher values of x/i lead to a higher probability of generating a tree-like network. The generated networks are always single output.

Composed: Randomly compose a few small-sized DAG topologies to form a larger network. This method is parameterized by the lower y_l and upper y_h bounds of the size of DAG topologies, the starting number of components c_0, the starting number of PIs x_0, the number of networks z of the same configuration to generate, the increment of the number of PIs Δx and of the number of components Δc. Upon initialization, the generator enumerates all isomorphic DAG topologies of y_l to y_h vertices. Then, it starts by generating networks of $x = x_0$ PIs and composed of $c = c_0$ components and keeps a counter of how many networks have been generated. After generating z networks, the values of x and c are increased by Δx and Δc, respectively. Given the current values of x and c, a network is generated by:

1. Create x PIs.
2. Randomly choose a topology T from the list.
3. For each hanging input of T, randomly decide on an existing node (a PI or a node in a created component) to connect to.
4. For each vertex in T, randomly decide on a gate type.
5. For each edge in T, randomly decide whether it is complemented.
6. If the number of created components is smaller than c, repeat from step 2.
7. Assign all nodes without fanout to be POs.

6.4.2 Testcase Minimization

Assuming that the concerned defect is deterministic, there is a *core* in any given failure-inducing testcase, which is a subset of the testcase essential for observing the defect. The other parts of the network are said to be *irrelevant* for observing the defect and can be removed. For example, for a defect caused by the algorithm trying to insert an XOR gate into an AIG, which is interpreted as inserting an AND gate instead, a core in the testcase may be a subnetwork computing the XOR function. Due to the localized-computation design style of modern scalable logic synthesis algorithms, the cores are usually small-sized. We say that a core is *minimal* if, for any node n, removing n results in never observing the defect again no matter how the fanins and fanouts of n are re-connected. A minimal core in a failure-inducing testcase may or may not be unique. The goal of testcase minimization is to find a minimal core in a given failure-inducing testcase.



(a) *Remove PI*: The TFO of n is simplified.

(b) *Remove PO*: The MFFC of n is removed.

(c) *substitute gate*: The MFFC of n is removed and the TFO of n is simplified.

(d) *Simplify TFO*: The TFO of n is simplified.

(e) *Remove MFFC*: The MFFC of n is removed.

(f) *Remove gate*: Only n is removed.

Figure 6.1: Illustration of the reduction stages.

We develop a testcase minimization tool for logic networks similar to delta debugging but without adopting binary search. Given a network and an AUT with verification (i.e. an oracle), our testcase minimizer iteratively tries to reduce the network and tests if the defect is still observed. Only the reduction operations that preserve observing the defect are kept; otherwise, the operation is undone. Different reduction operations are tried in six stages with increasing (finer) granularity as follows:

(a) *Remove PI*: Substitute a PI n with constant zero, thus simplifying its TFO by constant propagation. In AIGs, some nodes in the TFO of n that are connected to n without complementation are removed, and so are their MFFCs.

(b) *Remove PO*: Substitute a PO n with constant zero, thus removing its MFFC.

(c) *substitute gate*: Substitute a gate n with constant zero, thus removing its MFFC and simplifying its TFO by constant propagation (as in (a)).

(d) *Simplify TFO*: Assign fanins of a gate n as new POs, and then substitute n with constant zero. This operation is less aggressive than the previous one because only the TFO of n is simplified and its MFFC is kept.

(e) *Remove MFFC*: Substitute a gate n with a new PI. This operation does not cause constant propagation in its TFO and only removes the MFFC of n.

(f) *Remove gate*: Assign fanins of a gate n as new POs, and then substitute n with a new PI or with one of its fanins. Only n is removed.

Figure 6.1 illustrates the effects of an operation in each of the reduction stages, where small triangles at the bottom and on top are PIs and POs, respectively, and circles are specific nodes. Regions filled in blue are removed after the operation, and regions marked in yellow are simplified by constant propagation after the operation. Wires and PIs or POs drawn in green are added after the operation.

The relative granularity of stages *remove PI* and *remove PO* depends on the shape of the network. For networks with smaller TFO of PIs and less logic sharing in the TFI of POs, *remove PO* reduces the network faster; for networks with smaller MFFC of POs and more reconvergences near the PIs, *remove PI* reduces the network faster. Thus, the first stage to apply is heuristically decided by whether the network has more PIs than POs (*remove PO* is applied first) or more POs than PIs (*remove PI* is applied first).

In each stage, the minimizer backs up the current network, randomly samples a PO or a gate as n, and performs the corresponding reduction operation. If the defect is not observed anymore after reduction, the backup is restored. This procedure is repeated until all POs or all gates have been sampled, or until a pre-defined number of operations have been tried.

The resulting network after minimization cannot be reduced anymore because the last stage tries every possibility to remove one gate. Thus, by definition, the minimized testcase is guaranteed to be a minimal core. However, minimal cores are not necessarily unique, so it is possible that a different order of reduction operations (e.g. by using a different random seed) results in a smaller minimal core.

The minimized testcases are, in most cases, highly destructed and cannot be recognized or reverse-engineered anymore. Therefore, the testcase minimizer does not only facilitate the debugging process but also the communication between developers when commercially-sensitive benchmarks are involved.

6.4.3 Usage Example

The testing and debugging toolkit described in this chapter is implemented in *mockturtle*[5] as part of the EPFL open-source logic synthesis libraries [Soe+22]. The toolkit supports testing and debugging any application that takes a logic network, written in AIGER (for AIGs) or Verilog (for other network types supported in *mockturtle*, such as XAGs and MIGs) formats, as input.

Figure 6.2 shows an example workflow of our toolkit. In this example, the toolkit is used to fuzz test an algorithm implemented in *mockturtle* (marked in green), and then, if a defect is

[5]Available: https://github.com/lsils/mockturtle

```
#include <mockturtle/mockturtle.hpp>
using namespace mockturtle;

int main()
{                                                       Oracle
  auto opt = [](aig_network aig) -> bool {
    aig_network const aig_copy = aig.clone();
    aig_resubstitution(aig);
    aig_network const miter = *miter(aig_copy, aig);
    return *equivalence_checking(miter);
  };
                                                        Fuzzer
  fuzz_tester_params fuzz_ps;
  fuzz_ps.file_format = fuzz_tester_params::aiger;
  fuzz_ps.filename = "fuzz.aig";
  fuzz_ps.timeout = 20; // 20 minutes
  auto gen = random_aig_generator();
  network_fuzz_tester<aig_network, decltype(gen)>
    fuzzer(gen, fuzz_ps);
  bool has_bug = fuzzer.run(opt);

  if (!has_bug) return 0;
                                                        Minimizer
  testcase_minimizer_params min_ps;
  min_ps.file_format = testcase_minimizer_params::aiger;
  min_ps.init_case = "fuzz";
  min_ps.minimized_case = "fuzz_min";
  testcase_minimizer<aig_network> minimizer(min_ps);
  minimizer.run(opt);

  aig_network aig;
  lorina::read_aiger("fuzz_min.aig", aiger_reader(aig));
  write_dot(aig, "fuzz_min.dot");
  std::system("dot -Tpng -O fuzz_min.dot");

  return 0;
}
```

Figure 6.2: Example code to use the proposed toolkit to generate, minimize, and visualize a failure-inducing testcase.

```
auto opt = [](std::string filename) -> std::string {
  return "abc -c \"read " + filename + "; rewrite\"";
};
```

Figure 6.3: Example code to use the toolkit for testing and debugging an external tool, ABC.

observed, minimizes the generated failure-inducing testcase (marked in red). This can be done similarly for other C++-based tools that include *mockturtle* as a library.

Our toolkit is also applicable for testing and debugging external tools. In this case, the lambda function in lines 6 to 11 in Figure 6.2 shall be replaced by one that resembles the code in Figure 6.3.

Similar to `aigfuzz` and `aigdd`, calling the oracle as a system command requires switching the program control through the command shell and interfacing the testcases by reading and writing files. With the possibility of a tight integration as in Figure 6.2, these interfacing overheads can be eliminated and thus, empirically, making the automated testing and debugging workflow about 10× faster.

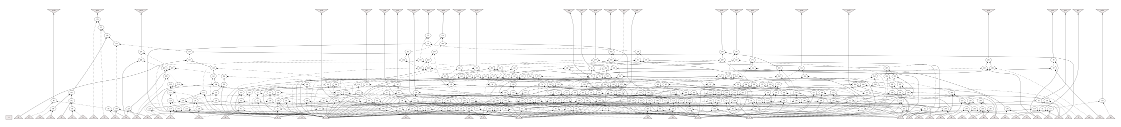
6.5 Case Study

As a case study, we apply the toolkit on a known defect in a variation of *cut rewriting*, which uses a compatibility graph to identify compatible substitution candidates [Rie+19a], implemented in *mockturtle*.[6] The defect can be observed by having a cyclic network after applying the algorithm. The failure-inducing core of this defect is shown in Figure 6.4 (d). The cyclic result is caused by the algorithm observing $n_7 \oplus n_2$ as a substitution for n_{11} and $n_{11} \oplus n_2$ as a substitution for n_7, and trying to apply the two substitutions at the same time. To identify that the two substitution candidates are in conflict, the algorithm should check, for every pair (A, B) of candidates, if the root of A is contained in the cut of B and the root of B is contained in the cut of A. This would be a feasible fix for the defect but would impact the efficiency of the algorithm. Another rewriting algorithm that does not use the compatibility graph but eagerly substitutes each candidate before searching for the next one is available in *mockturtle*.[7] However, when not affected by the defect, the defective algorithm has on average better quality of result than eager rewriting. Also, the defect seems to be observed very rarely, as will be discussed in Section 6.5.1. As a compromise, both algorithms are kept in *mockturtle*.

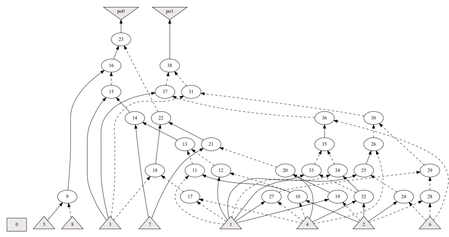
The first reported failure-inducing testcase for this defect is shown in Figure 6.4 (a). The original testcase was not minimized by the reporter and have 49 PIs, 272 AND gates, and 28 POs. It took a human expert about 30 minutes to manually reduce the testcase to Figure 6.4 (d), with 3 PIs, 8 gates, and 2 POs. Using the testcase minimizer, the original testcase is minimized to the same graph (subject to permutations of the two POs) within a second and using 94 oracle calls. In Section 6.5.2, we study the effectiveness and necessity of the reduction stages described in Section 6.4.2.

[6]The function `cut_rewriting_with_compatibility_graph` can be found in `algorithms/cut_rewriting.hpp`.

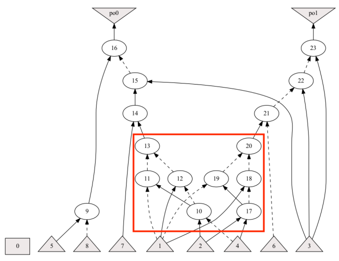
[7]The function `cut_rewriting` can be found in the same header file.



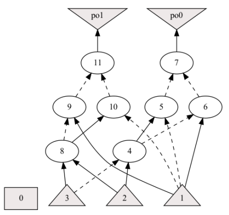
(a) Before reduction, the original testcase is too big for human eyes to understand.



(b) *Remove PO* only.



(c) *Remove PO* and *substitute gate*.



(d) The minimum failure-inducing testcase.

Figure 6.4: The failure-inducing testcase for an algorithm implemented in *mockturtle* and intermediate results of minimizing it.

6.5.1 Capturing The Defect with Fuzz Testing

Using AIGs

Knowing the existence of the defect, we investigate if our fuzz tester is capable of generating another failure-inducing testcase. However, even though the code line coverage has reached its maximum (100% excluding the lines disabled by the algorithm's options), the defect is not observed with more than a billion (10^9) regular (i.e., without leveraging knowledge of the known core) fuzz tests. Even if we limit the sampling space to the 3-input, 8-gate, 2-output topology as in Figure 6.4 (d) and leaving only the connections to PIs and edge complementations as random choices, there are still $6^2 \times 3^4 \times 2^{16} = 191\,102\,976$ different possible networks, out of which only $3! \times 2^3 = 48$ networks (equivalent to Figure 6.4 (d) subject to permutation and negation of PIs) are failure-inducing.

This case evidences that rare corner-case defects exist in logic synthesis applications, and the identification of them may only rely on real-world benchmarks. In these cases, the testcase minimization techniques are important to automatize the extraction of the failure-inducing core, which facilitates communication and debugging.

Using XAGs

We observe that the XOR functions in the core (nodes 9, 10, 11 and nodes 5, 6, 7 in Figure 6.4 (d) are necessary. Using any of the randomized methods described in Section 6.4.1, the possibility of generating an XOR function composed of three AIG nodes is low. However, it is much more likely to generate an XOR gate in an XAG. As the implementation is generic and works for both

Table 6.1: Fuzzing the defective cut rewriting with XAGs.

Method	#Tests	Time (s)
Random	8150	1.8
Topology	44498	6.6
Composed	77573	22.8

AIGs and XAGs, we can try to capture the defect using XAGs instead. Table 6.1 shows that all the three methods successfully capture the defect within reasonable runtime.

6.5.2 Effects of The Reduction Stages

Given the initial failure-inducing testcase as in Figure 6.4 (a), using the default settings, our testcase minimizer produces the minimal failure-inducing testcase as in Figure 6.4 (d), which is a 97% reduction rate in gate count. The minimality can be proved by trying to remove each gate and seeing that any possible resulting testcases are not failure-inducing.

Figures 6.4 (b) and 6.4 (c) show the reduction results if only some reduction stages are applied. The first stage, *remove PO* (*remove PI* is skipped because there are more PIs than POs), provides already 89% reduction of the testcase by removing large cones of irrelevant logic and quickly concentrates to the transitive fanin cone of two POs (Figure 6.4 (b), 30 gates). The next stage, *substitute gate*, further reduces the size to 15 gates (Figure 6.4 (c)), and the failure-inducing core is easily observable (marked with a red box). However, the other nodes on top of the core cannot be removed in this stage because substituting any of them with constant zero also removes part of the core. This can be accomplished by adding the stage *simplify TFO*, resulting in Figure 6.4 (d). The two key operations are adding PO at nodes 13 and 20 and substituting nodes 14 and 21 with constant zero. It is also possible to reach the minimum by adding only the stage *remove gate*, but it requires at least 6 operations to remove nodes 14, 15, 16, 21, 22 and 23 one by one, showing that this stage operates in finer granularity. It may seem that the stage *remove MFFC* is not necessary. However, this is only because the failure-inducing core in this example does not have irrelevant transitive fanin gates (i.e., it is connected to PIs) in the original testcase. When this is not the case, the stages *remove MFFC* and/or *remove gate* are necessary to obtain the minimum.

6.6 Experimental Results

6.6.1 Fuzzing Open-Source Logic Synthesis Tools

To demonstrate the effectiveness of fuzzing and compare different testcase generation methods, we fuzz-tested the following open-source logic synthesis tools: *mockturtle*[8] [Rie+19b],

[8]Available: https://github.com/lsils/mockturtle. Commit `cf4769f`.

Table 6.2: Fuzz testing results.

		aigfuzz #Tests = 1000			Random #Tests = 1000			Topology #Tests = 5000			Composed #Tests = 5000		
	AUT	#FITs	Size	Time	#FITs	Size	Time	#FITs	Size	Time	#FITs	Size	Time
mockturtle	aig_resub	0	-	10.3	0	-	3.5	0	-	0.02	1	14.0	0.03
	sim_resub	3	812.3	22.3	0	-	4.2	6	5.0	0.06	93	21.7	0.11
ABC	if -u	952	2476.0	32.5	1000	950.0	14.6	1716	4.7	12.8	3749	20.3	13.0
	mfs -dael	956	2515.1	4.5	969	978.8	1.6	0	-	14.2	1047	23.9	12.3
	mfsd	473	3935.9	30.5	584	1078.4	14.1	0	-	14.1	120	24.2	14.1
	mfsd -cd	481	3959.7	130.0	73	1266.4	47.8	0	-	14.2	1	20.0	14.4
	mfse	458	2763.3	14.7	93	940.3	13.6	1	5.0	16.0	1056	23.4	15.0
	stochsyn	13	1164.8	14.6	0	-	6.0	0	-	12.5	14	18.9	12.5
LSOracle	aigscript	1	7364.0	38.1	0	-	16.4	0	-	32.8	1	14.0	32.8
	deep	12	3227.3	41.6	0	-	21.5	0	-	33.3	6	23.0	32.8
	xmgscript	3	1056.0	21.6	2	350.0	10.5	38	4.9	11.5	99	20.1	12.0
	Average		2941.6	32.8		995.5	14.0		4.7	14.7		21.5	14.5

`aig_resub = mockturtle::aig_resubstitution`, `sim_resub = mockturtle::sim_resubstitution`
`if -u = abc> "bms_start; if -u; strash"`, `mfs -dael = abc> "&if; &mfs -dael; &st"`
`mfsd = abc> "&mfsd; &st"`, `mfsd -cd = abc> "&mfsd -cd; &st"`
`mfse = abc> "if; mfse; strash"`, `stochsyn = abc> "&stochsyn resub"`
`aigscript = lsoracle> aigscript`, `deep = lsoracle> deep`, `xmgscript = lsoracle> xmgscript`

Table 6.3: Testcase minimization results.

	Original	aigdd			Ours		
AUT	Size	Size	#Calls	Time	Size	#Calls	Time
`mockturtle::cut_rewriting_with_compatibility_graph`	272	8	210	32.5	8	96	0.1
`mockturtle::sim_resubstitution`	615	7	735	11.6	8	351	2.5
`abc> &mfsd -cd; &st`	1050	31	1198	150.0	20	857	120.5
`abc> if; mfse; strash`	1850	6	834	61.2	5	333	30.3
`abc> &stochsyn resub`	3228	10	1124	59.6	8	411	21.1

ABC[9] [BM10], and *LSOracle*[10] [Net+19]. Table 6.2 lists the commands or functions where defects have been observed. Fuzz testing campaigns were conducted on each AUT using `aigfuzz` and the three network generation methods described in Section 6.4.1. In each campaign, `aigfuzz` and the method Random ran 1000 tests, whereas the methods Topology and Composed ran 5000 tests. In Table 6.2, column #FITs lists the total number of failure-inducing testcases generated, column Size lists the average size (number of gates) of the failure-inducing testcases, and column Time lists the total runtime in minutes including the oracle calls.

The Composed method captured defects in all of the listed AUTs. On average, Composed is about 2× faster than `aigfuzz` and it tests on 5× more networks. This is because the Composed testcases are, on average, 7% in size compared to those generated by `aigfuzz`. Also, notice that for AUTs in *mockturtle*, the runtimes of our fuzzing methods are about 10× faster than `aigfuzz` thanks to the tight integration.

6.6.2 Testcase Minimization

We compare our testcase minimizer to `aigdd` using the user-reported failure-inducing testcase in Section 6.5 and four bigger testcases found by fuzz testing in Section 6.6.1. In Table 6.3, column Size lists the number of gates of the original and the minimized testcases, column #Calls lists the number of oracle calls and column Time lists the total runtime in seconds. It can be observed that our minimizer reduces the testcases into minimal cores of roughly the same or smaller sizes compared to `aigdd`, using on average 50% oracle calls and 50% runtime.

6.7 Discussions

6.7.1 Non-deterministic Defects

Non-deterministic defects may be hard to debug because they cannot always be reproduced. Non-determinism may come from a random number generator without a fixed seed, a race

[9]Available: https://github.com/berkeley-abc/abc. Commit `31519bd`.
[10]Available: https://github.com/lnis-uofu/LSOracle. Pull request #81.

condition in concurrent computation, or accessing to uninitialized or unintended (index-out-of-bounds) memory. If a non-deterministic defect is first observed with a large testcase, it may be difficult to minimize it while maintaining the defect being observed. In such cases, fuzz testing may help generate smaller testcases to observe the defect deterministically.

6.7.2 Other Applications of The Toolkit

In addition to testing and debugging, the proposed tools can also be used for finding examples with specific properties. For example, an open problem in logic synthesis is whether it is better to heavily optimize an AIG before transforming into MIG, or to perform optimization directly with an MIG. Our toolkit can be used to generate minimal examples where one optimization script obtains better results than the other, which might help researchers identify weaknesses in the algorithms.

6.8 Future Directions

Our network fuzzer currently does not support generating k-LUT networks easily without specifying all possible LUT functions as different gate types. This can be mitigated by integrating a random truth table generator.

In addition to minimizing the failure-inducing input networks, when the defective AUT involves multiple independent algorithms (i.e., a *script* with a sequence of *commands*), it would also be helpful to minimize the script and remove irrelevant commands. This can be accomplished by automatic binary search, similar to delta debugging.

6.9 Summary

In this chapter, we survey automated testing and debugging techniques and provide an open-sourced toolkit specialized for gate-level logic synthesis applications. While random fuzz testing can already catch many higher-frequency defects, the topology-based fuzzing methods provide a more systematic approach to thoroughly test topology-related corner cases. After failure-inducing testcases are found, the testcase minimizer can be used to reduce their size efficiently to facilitate manual debugging (and also anonymizing sensitive testcases). Moreover, our testcase minimization technique guarantees finding a minimal core in the failure-inducing testcase, which often gives insights into the cause of the defect and may also be used to categorize testcases for the same AUT. The case study shows that (1) some defects may be difficult to catch by fuzz testing, thus testcase minimization is important when we need to rely on real-world testcases; and (2) testing with more functionally-compact networks, such as XAGs, may help to detect some defects in generic logic synthesis algorithms.

AQFP Circuit Optimization Part II

7 Adiabatic Quantum-Flux Parametron

Adiabatic quantum-flux-parametron (AQFP) is an emerging *superconducting electronic* (SCE) technology receiving increased interest. Featuring an ultra-low energy consumption and a high switching speed, AQFP is a promising and attractive alternative to CMOS-based digital families for high-performance computing.

The AQFP technology imposes some special constraints uncommon to classical CMOS technologies. First, because every gate in an AQFP circuit is clocked, all input signals for a logic gate must arrive at the same time (in the same clocking phase). To ensure this, shorter data paths need to be delayed with clocked buffers. Moreover, the output signal of AQFP logic gates cannot be directly branched to feed into multiple fanouts. Instead, splitters are placed at the output of multi-fanout gates to amplify the output current, and they are also clocked. Special care needs to be taken in EDA to fulfill these constraints (i.e., to *legalize* the circuit for AQFP), which is the main topic in Part II of this thesis. Besides, the elementary logic gate in AQFP is the majority gate and input negation is for free, thus making MIGs a natural choice for AQFP logic synthesis. In Chapter 10, we will integrate methods developed in Part I for MIGs with new algorithms proposed in Part II for AQFP legalization to form a complete AQFP logic synthesis flow.

In this chapter, we introduce the basic concepts related to SCE and AQFP, including the gate-level and architecture-level clocking schemes, as well as the special design constraints. We will also define the mathematical abstraction and terminology to be used in the remainder of this thesis.

7.1 Superconducting Electronics

Magnetic Flux Quantum

The magnetic flux threading a superconducting material is, in contrast to normal conductors, quantized. The *magnetic flux quantum*, denoted as Φ_0, is the smallest unit of magnetic flux for any superconductor. Its value, $\Phi_0 = \frac{h}{2e} \approx 2.07 \times 10^{-15}$ Wb, is a constant that can be derived from the Planck constant h and the electron charge e.

Josephson Junction

Josephson junction (JJ) is the active device in superconducting circuits. It is composed of two superconducting regions and a weak link with no or weakened superconductivity between them. The number of JJs in a superconducting circuit is related to its energy consumption and complexity. Thus, JJ count is commonly used as the cost metric for superconducting circuits.

The characteristic parameter associated with a JJ is its *critical current* I_c, which is the maximum current amplitude through the JJ. The current and voltage across a JJ are related to its *Josephson phase* φ, which is the phase difference of the wave functions of Cooper pairs in the two superconductors of the JJ. Known as the *Josephson effect*, the current flowing through a JJ is related to its phase by $I(t) = I_c \sin\varphi(t)$ and can flow for indefinitely long without dissipation.

Superconducting Digital Computing

Superconductors can be used in both classical digital logic and quantum computing. We focus on the former in this thesis. As the underlying computing paradigm remains the same, that is, explainable using Boolean logic instead of superpositioned quantum qubits, existing logic synthesis techniques can be easily applied. There are currently two main families of SCE technologies, namely *single-flux quantum* (SFQ) [LS91b] and *adiabatic quantum-flux parametron* (AQFP) [Tak+13]. Both of them (still) require a cryogenic environment for their correct operation. Nevertheless, even with the refrigerating cost taken into account, SCE technologies still achieve significantly lower energy consumption compared to the CMOS family.

7.2 Basic Principles of Adiabatic Quantum-Flux Parametron

7.2.1 Parametron and Quantum-Flux Parametron

The *parametron*, proposed by Goto in 1954, was a candidate of the logic component in computers competing against the transistor before the breakthrough in semiconductor technology had made the latter become a much more reliable and economical choice. The parametron is essentially a resonant circuit utilizing the parametric oscillation phenomenon. By applying an alternating excitation current of frequency $2f$ to a balanced system, an oscillation of frequency f is generated and it is stable in either of two phases differing by π radians. The two stable points are thus used to represent logic 0 and 1 [Got59].

The *quantum-flux parametron* (QFP) uses JJs in the parametron circuit to create persistent current. The circuit schematic of the basic QFP model is shown in Figure 7.1. When the

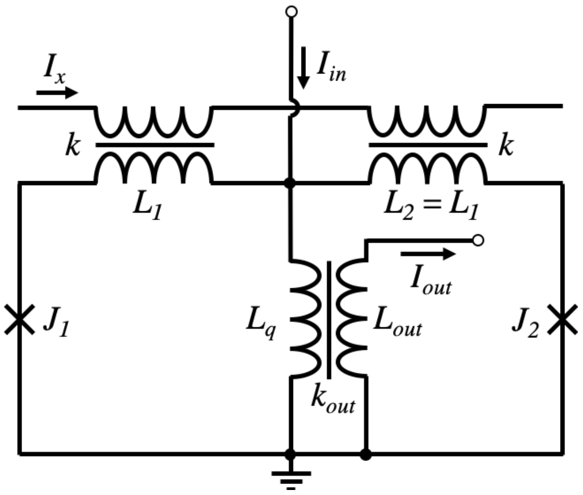


Figure 7.1: Circuit schematic of the QFP.

excitation current I_x is applied, which creates an excitation flux in the loops by inductive coupling, the potential energy of the QFP appears to have two local minima at a positive value and a negative value of output flux (Figure 2 in [Hos+91] and Figure 2 (a) in [Tak+13]). With a small input current, the system falls into either one of the stable states, determining the direction of the output current [Har+87; Hos+91].

7.2.2 Adiabatic Operation

The AQFP is a QFP circuit operated in the adiabatic mode. The term *adiabatic* in the name of AQFP refers to switching operations without, or with very low, loss or gain of electronic charge. By carefully tuning the circuit parameters in a QFP gate, it has been shown that the switching energy dissipation of an AQFP gate can be reduced to much lower than (i.e., 12% of) $I_c\Phi_0$, which is the limit for any technology in the SFQ family [Tak+13]. The AQFP switching energy dissipation is close to the theoretically predicted limit[1] [KL70].

In short, by operating in the superconductive region, AQFP circuits achieve zero static energy dissipation [Har+87]; by operating in the adiabatic mode, AQFP circuits achieve very small dynamic energy consumption [Tak+13].

7.2.3 Logic Computation

In AQFP digital circuits, logic ‘0’ and ‘1’ are represented by different current directions of the same magnitude, which is a result of a quantum flux existing in one of the two loops (Figure 7.1), instead of low and high voltages as in CMOS. The basic circuit components in

[1]The typical value of I_c is about 50 μA, hence $I_c\Phi_0$ is of order 10^{-19} J. On the other hand, $k_B \approx 1.4 \times 10^{-23}$ is the Boltzmann constant, and the operation temperature T is typically several Kelvin, making the theoretical limit $k_B T$ at the order of 10^{-22} J.

AQFP logic are the buffer cell (as shown in Figure 7.1, using 2 JJs) and the branch cell (a current forking circuit, without any JJs). A NOT gate is created with a buffer with negative inductive coupling ($k_{out} = -k$). A majority-3 (MAJ3) gate can be constructed by combining three buffer cells with a reverted branch cell (i.e., a 3-to-1 merger). Other preliminary logic gates, such as the AND2 and OR2 gates, can be built from the MAJ3 gate with a constant input (constant 0 for AND2 and constant 1 for OR2) made of an asymmetric buffer cell. Input negation of logic gates is realized using a negative mutual inductance and is of no extra cost [TYY15]. Like other superconducting technologies, the commonly-used cost metric for AQFP circuits is the JJ count. A buffer costs 2 JJs, a branch cell is of zero JJ-cost, and a logic gate based on MAJ3 costs 6 JJs [TYY15].

7.2.4 Gate-level Clocking Schemes

Logic gates, buffers, and splitters in AQFP are periodically activated and reset by an alternating excitation current [Tak+13]. A gate takes its inputs, computes its logic function, and provides its output with the presence of the excitation current. In the absence of the excitation current, an AQFP gate produces no output current (i.e., neither logic ‘0’ nor logic ‘1’). Thus, two cascaded gates must be fed with consecutive clocking phases, where the capturing gate is activated later than, but overlapping with, the activation of the launching gate, such that the information can be propagated along the circuit. Using similar terminology as in logic synthesis, we call the capturing gate a *fanout* of the launching gate, and the launching gate a *fanin* of the capturing gate.

Various clocking schemes have been proposed. 3-*phase clocking* was used in earlier works [Tak+13; TYY15; Aya+17], where three excitation currents with a phase shift of 120° to each other are fed into different levels of gates. A few years later, 4-*phase clocking* was proposed [Tak+17] and has remained the most commonly-used clocking scheme until now. In 4-phase clocking, the phase shift decreases to 90°, the number of alternating current sources decreases to 2, and the number of clocking phases in each clock cycle increases to 4, allowing for slightly lower latencies by enabling a logical depth of 4 gates instead of 3 per cycle. In both 3- and 4-phase clocking, logic gates in each level are assigned to one of the three or four phases and *phase synchronization* must be ensured: Any fanin of a gate g must be at the previous phase of g.

Another clocking scheme is *delay-line clocking* [Tak+19], where a single alternating excitation current is used and transmission lines are inserted between levels to delay the clock. Delay-line clocking not only allows for even lower latency but also enables the *phase-skipping operation* [SAY21; YTY21], reducing the number of path-balancing buffers.

In this thesis, we use p_{clk} to denote the number of phases in a (gate-level) clock cycle. Typically, $p_{\text{clk}} = 3$ or 4.

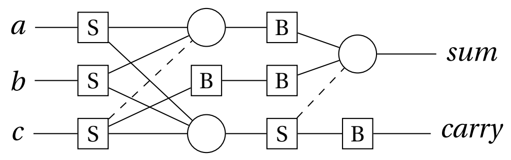


Figure 7.2: An AQFP-legalized full adder circuit.

7.3 AQFP Design Constraints

Logic gates in an AQFP circuit need to be activated and deactivated periodically by an excitation current [Tak+17]. In other words, every gate in an AQFP circuit is clocked, and all input signals have to arrive at the same clock cycle. To ensure this, shorter data paths need to be delayed by clocked buffers. Moreover, the output signal of AQFP logic gates cannot be directly branched to feed into multiple fanouts. Instead, splitters are placed at the output of multi-fanout gates to amplify the output current. A splitter cell is composed of a buffer cell and a 1-to-n branch cell (usually, $2 \leq n \leq 4$) and is also clocked. As the cost of splitters comes mostly from the buffer cells, in the remainder of this thesis, we do not distinguish buffers from splitters and model them using the same abstraction. Also, in all figures, we use circles to represent MAJ gates and squares to represent buffers/splitters.

To illustrate the AQFP technology constraints, Figure 7.2 shows a full adder as a legalized AQFP circuit. Splitters (S squares) are inserted to drive multiple fanouts and buffers (B squares) are used to balance all paths from a PI to a PO.

7.4 Memory Devices and Architectural Clocking

To implement sequential circuits using a similar finite-state-machine model as CMOS digital systems, AQFP memory devices are needed. At least two possible designs have been proposed in the literature: D-latch and QFPL-based NDRO.

A simplified AQFP *feedback delay latch* (D-latch) is depicted in Figure 7.3, where the 4-phase clocking scheme is used. A D-latch takes an *Enable* signal E and a *Data* signal D as inputs. Its operation is illustrated by the truth table shown in Table 7.1. When $E = 0$, the majority gate has input values $(0, 1, Q_n)$, thus keeping the same internal state $Q_{n+1} = Q_n$; when $E = 1$, the majority gate has input values (D, D, Q_n), thus the internal state is overwritten by the new data D [Tsu+17].

A *quantum-flux-parametron latch* (QFPL) is a special AQFP gate that can hold its state when the excitation current is low. The internal state of an QFPL is updated only when its two inputs A and B present the same value; otherwise, it keeps the previous state. Combining an QFPL and some logic gates, a *non-destructive-read-out* (NDRO) can be made, as shown in Figure 7.4. An NDRO also takes an *Enable* signal E and a *Data* signal D as inputs and has the same truth table as in Table 7.1. When $E = 0$, we have $A = 0$ and $B = 1$, thus the QFPL holds its previous

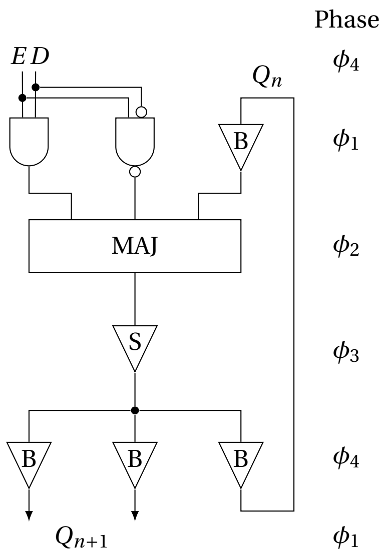


Figure 7.3: Circuit schematic of an AQFP D-latch.

Table 7.1: Truth table of D-latch and NDRO.

E	D	Q_{n+1}	Action
0	0	Q_n	Hold
0	1	Q_n	Hold
1	0	0	Write 0
1	1	1	Write 1

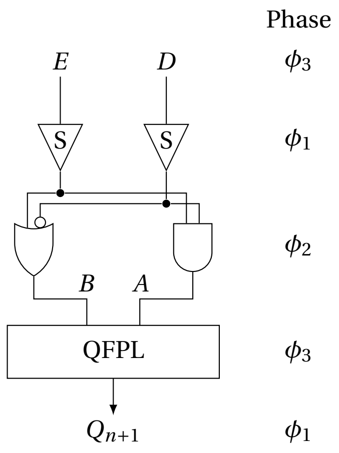


Figure 7.4: Circuit schematic of an QFPL-based NDRO.

state; when $E = 1$, then $A = B = D$ and the new data is written into the QFPL [Sai+21].

For a D-latch, an update to the state, caused by a new value at the input D enabled with $E = 1$, is propagated through the circuit and changes the output Q_{n+1} 4 phases later. In contrast, an update to the state of a NDRO is available at the output 3 phases later.

In a classical sequential circuit model, the data D inputs of registers come from the outputs of the previous-stage combinational circuit, the outputs Q of registers are connected to the inputs of the next-stage combinational circuit, and the enable E input of registers comes from an architectural clock (in contrast to the gate-level clock discussed in Section 7.2.4). In the CMOS paradigm, the enabling signal of registers is the rising edge or falling edge of a periodic clock signal. In contrast, in AQFP, the enabling signal E is kept at 0 most of the time and become 1 once every k gate-level clock cycles, where the value k depends on the length of the critical combinational path. In this thesis, we denote the number of phases in an architectural clock cycle by $p_{\text{arch}} = k \cdot p_{\text{clk}}$.

7.5 Abstraction and Terminology

In the remaining part of this thesis, we abstract AQFP circuits at the gate level as homogeneous logic networks using the same terminology as in Part I. Because the basic logic gate in AQFP is the majority gate and input negation is cost-free, AQFP logic networks are essentially MIGs.

Buffers and splitters need to be inserted in an AQFP logic network to fulfill technology constraints, producing a *mapped network*. A mapped network $N' = (V', E')$ is a network extended from a (unmapped) network $N = (V = I \cup O \cup G, E)$, where the node set V' is supplemented with a set B of *buffers*, i.e., $V' = I \cup O \cup G \cup B$. A buffer is a node with an in-degree 1, modeling an AQFP buffer cell (when having an out-degree 1) or an AQFP splitter cell (when having an out-degree larger than 1). In a mapped network, the definition of the fanouts of a gate is modified by ignoring any intermediate buffers, i.e., a path from a gate g to one of its fanouts

$g_o \in \mathrm{FO}(g) \subset (G \cup O)$ may include any number of buffers in B, but never another gate. The definition of fanins is modified similarly. The *fanout tree* of a gate (or a PI) n, denoted by $\mathrm{FOT}(n)$, is the set of buffers between n and any gate or PO in $\mathrm{FO}(n)$.

A *schedule* of a network is a function $\mathcal{S} : V \to \mathbb{Z}_{\geq 0}$ that assigns a non-negative integer $\mathcal{S}(n)$ to each node $n \in V$, called the *level* of n. A *valid* schedule must fulfill the condition that $\forall n \in V, \forall n_i \in \mathrm{FI}(n), \mathcal{S}(n_i) < \mathcal{S}(n)$. We do not consider invalid schedules in this thesis. The depth of a network $N = (V = I \cup O \cup G, E)$ with a schedule $\mathcal{S}$ is defined as $d(N) = \max_{o \in O} \mathcal{S}(o)$.

This chapter serves as the background introduction for Part II of this thesis. We introduced the basic principles and the special design constraints of the AQFP technology, which motivated the research to be done in the following chapters: In Chapter 8, we first discuss the necessity of these constraints and tradeoffs caused by possible relaxations. Then, in Chapter 9, we propose systematic legalization methods to fulfill the constraints. Finally, we put everything together as an AQFP logic synthesis and technology mapping flow in Chapter 10.

8 Impact of Sequential Design on AQFP Technology Constraints

8.1 Motivation

As switching energy dissipation in AQFP is related to the number of JJs, reducing the JJ count of AQFP circuits has been the primary optimization goal along with reducing circuit latency. This in turn, also helps to reduce the overall circuit area as AQFP primitives have a large footprint due to their output transformer. In previous works, the AQFP buffer and splitter insertion problem has been formulated as follows: All paths should be balanced to the same length (*path balancing*), and all gates, including primary inputs, with multiple fanouts must be branched (*fanout branching*). Surprisingly, research has found that a large portion of JJs in AQFP benchmark circuits is dedicated to buffering cells to fulfill these technology constraints.

It is very seldom the case in modern EDA where the design under synthesis is purely combinational without any memory devices. In conventional EDA flows, we usually cut off combinational parts of the circuit for logic synthesis because the combinational optimization problem is simpler than the sequential one. However, in the context of AQFP legalization, it is important to understand the mechanism of the sequential model when formulating the constraints and diving into solving the legalization problem, because the required constraints are not exactly the same in a purely combinational scenario and in a sequential design.

While the path-balancing and fanout-branching constraints are absolutely required for the correct operation of an AQFP combinational[1] circuit without memory devices, in the context of a sequential computing model where combinational inputs and outputs are connected to registers, these constraints may be too conservative. According to the architectural clocking scheme currently used in AQFP sequential circuits, registers generally hold their values throughout the architectural clock cycle and their outputs can be taken by the next-stage combinational circuit multiple times. In other words, the same computation is repeated in waves in an AQFP combinational circuit. With a careful analysis, we argue that it is not always necessary to balance all paths to equal length. Instead, aligning the gate-level clock phases is

[1]Although AQFP gates are clocked, we use the terms *combinational* and *sequential* here in a similar sense as in CMOS digital circuits, considering the (architectural) clock connected to registers.

Table 8.1: Parameters involved in AQFP constraint formulation.

Parameter	Meaning	Reasonable value(s)
s_b	Buffers' splitting capacity (maximum number of fanouts)	$s_b \geq 1$, usually 3 or 4
s_i	PIs' splitting capacity (maximum number of fanouts)	$1 \leq s_i < s_b$ (See Section 8.2.2)
s_g	Gates' splitting capacity (maximum number of fanouts)	1
p_{clk}	Clocking scheme (number of phases in a gate-level clock cycle)	3 or 4
Φ_{ro}	Set of phase differences a register may produce its output relative to its input phase	$\Phi_{\text{ro}} = \{4\}$ or $\Phi_{\text{ro}} = \{3,4,5\}$ (See Section 8.2.2)

enough.

In this chapter, we discuss how architectural clocking and register design affect AQFP technology constraints. We argue that the commonly adopted constraint formulation is sometimes too conservative and propose relaxations to the constraints. Consequently, we also investigate how the relaxation of constraints affects the number of buffers needed, and discuss possible trade-offs when the constraints are relaxed.

8.2 AQFP Design Constraints

In most existing works related to AQFP technology legalization, the path-balancing and the fanout-branching constraints are assumed, which are mathematically defined as follows for a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$ and its associated schedule $\mathcal{S}$, subject to the *splitting capacities* (the maximum number of fanouts a node may have) $s_i = 1, s_g = 1$ and $s_b \geq 1$ of PIs, gates, and buffers, respectively.

- Path balancing: N' is *path-balanced* if

$$\forall n_i, n_o \in V' : (n_i, n_o) \in E' \Rightarrow \mathcal{S}(n_i) = \mathcal{S}(n_o) - 1, \tag{8.1}$$

$$\forall i \in I : \mathcal{S}(i) = 0, \text{ and} \tag{8.2}$$

$$\forall o \in O : \mathcal{S}(o) = d(N'). \tag{8.3}$$

- Fanout branching: N' is *properly-branched* if every PI has an out-degree no larger than s_i, every gate has an out-degree no larger than s_g, and every buffer has an out-degree no larger than s_b.

In particular, the path-balancing constraint has its origin in the phase synchronization require-

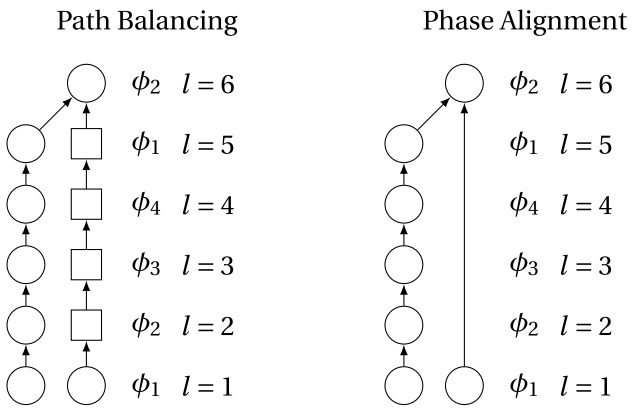


Figure 8.1: Illustration of path balancing and phase alignment. ($p_{\text{clk}} = 4$)

ment: an AQFP gate can only compute and output its logic function correctly at a gate-level clock phase ϕ_i if all of its fanins output their values at the previous phase ϕ_{i-1}. However, recall that the AQFP gate-level clock phases are not infinite but loop from ϕ_1 to $\phi_{p_{\text{clk}}}$ and then back to ϕ_1. In a 4-phase clocking scheme ($p_{\text{clk}} = 4$) for example, the next phase of ϕ_4 is ϕ_1, and the phase 5 phases after ϕ_4 is also ϕ_1. In other words, any phase difference of $k \cdot p_{\text{clk}} + 1$ shall be allowed, as illustrated in Figure 8.1. Moreover, it may be possible for the memory devices to output their values at more than one phase, depending on their circuit design.

Thus, we define the phase alignment constraint formally as follows, subject to two parameters: the clocking scheme p_{clk} (the number of phases in a gate-level clock cycle) and Φ_{ro} (the set of phase differences a register may produce its output relative to its input phase).

- Phase alignment: N' is *phase-aligned* if

$$\begin{aligned}
&\forall n_i, n_o \in V' : (n_i, n_o) \in E' \\
&\quad \Rightarrow \mathcal{S}(n_i) \bmod p_{\text{clk}} = (\mathcal{S}(n_o) - 1) \bmod p_{\text{clk}} \land \mathcal{S}(n_o) > \mathcal{S}(n_i), && (8.4) \\
&\forall i \in I : \exists \phi_i \in \Phi_{\text{ro}}, \mathcal{S}(i) \bmod p_{\text{clk}} = \phi_i \bmod p_{\text{clk}} \land \mathcal{S}(i) \geq \phi_i, \text{ and} && (8.5) \\
&\forall o \in O : \mathcal{S}(o) \bmod p_{\text{clk}} = 0. && (8.6)
\end{aligned}$$

In the following, we discuss which subset of these properties shall be required as AQFP technology constraints and the values of the parameters involved.

8.2.1 Phase Alignment Instead of Path Balancing

Existing works on AQFP sequential architectural design [Aya+21; Sai+21], logic synthesis [Xu+17; Cai+19c; Aya+20; Tes+21; MRM21; Meu+22], and buffer insertion-optimization [Hua+21; LRD22b; CD23; Fu+23a] conventionally adopt a conservative set of constraints: path balancing and fanout branching. Notice that fulfilling path balancing, with an additional constraint that $d(N') \bmod p_{\text{clk}} = 0$, implies fulfilling phase alignment with $\Phi_{\text{ro}} = \{0\}$. While this ensures correct and robust operation of the AQFP circuit even with fast clock frequencies, enforcing these constraints often leads to bulky circuits with more than half — sometimes up to 90% — area taken by buffers. In this section, we argue that in the context of synthesizing combinational logic between register stages, assuring phase alignment, instead of the stronger path-balancing constraint, is enough.

In [Sai+21], the authors proposed that when registers in several sequential stages share the same enable signal, which arrives once per p_{arch} phases matching the depth of the deepest stage, shallower stages do not need to be balanced to the same length as the deepest stage. The main reason is that memory devices output their value every p_{clk} phases and do not change their internal state for the entire architectural clock cycle until the enable signal arrives. Thus, although shallower stages finish their computation earlier than when the registers are enabled to take the next values again, the same computation is repeated every (gate-level) clock cycle, and the same computational results are produced repeatedly until the registers are enabled again to accept them.

With a similar reasoning, we extend the argument further and propose that the path-balancing constraint can be relaxed to phase alignment, formally stated as follows.

Lemma 8.1. *In an AQFP sequential circuit, let d be the longest path length between any two register stages, ϕ_{ro} be the phase difference between the register output Q_{n+1} and inputs D, E. Suppose that the register enable signal E is* 1 *for one phase in every $p_{arch} = k \cdot p_{clk}$ phases, where $p_{arch} \geq \phi_{ro} + d$, then fulfilling the phase-alignment constraint (Equations* (8.4) *to* (8.6)*), in addition to fanout branching, is enough to ensure correct sequential operation of the circuit.*

Proof. Without loss of generality, consider the computation propagated from one register stage I, through a combinational circuit N, to the next register stage O, in one architectural clock cycle. Suppose that $E = 1$ at time $t = 0$ and at time $t = p_{\text{arch}}$ (the unit of time is number of phases) and that $E = 0$ all the other time. Let the (multi-input, multi-output) Boolean function computed by N be f_N and let the values presented at the outputs of registers I at time $t = \phi_{\text{ro}}$ be $\vec{x}$, we will prove that the values presented at the inputs of registers O at time $t = p_{\text{arch}}$ are exactly $f_N(\vec{x})$.

First, observe that the same $\vec{x}$ is produced at I every p_{clk} phases until (excluding) $t = p_{\text{arch}} + \phi_{\text{ro}}$, i.e., at

$$t = \phi_{\text{ro}}, \phi_{\text{ro}} + p_{\text{clk}}, \phi_{\text{ro}} + 2 \cdot p_{\text{clk}}, \ldots, \phi_{\text{ro}} + (k-1) \cdot p_{\text{clk}}. \tag{8.7}$$

Comparing Section 8.2.1 against Equation (8.5), we conclude that for all combinational inputs i, its value is ready at time $t = \mathcal{S}(i)$ corresponding to its assigned level, as well as every p_{clk} phases afterward.

Next, consider a gate n with two fanins[2] n_{i1} and n_{i2} and suppose that the values of n_{i1} and

[2]We consider two fanins in the analysis for convenience, but the argument can be extended to any number of

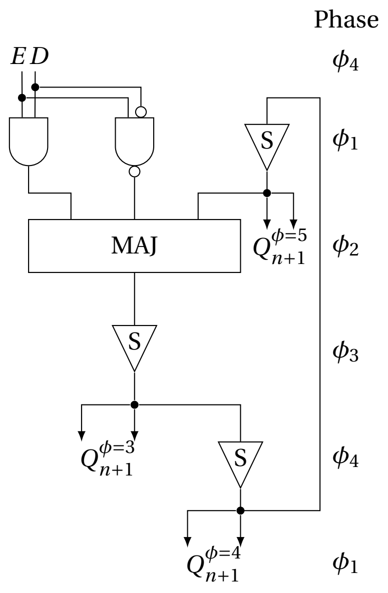


Figure 8.2: Circuit schematic of an improved D-latch design.

n_{i2} are ready at times corresponding to their assigned level, as well as every p_{clk} phases after these times, i.e., $t = \mathcal{S}(n_{i1}) + j \cdot p_{\text{clk}}$ and $t = \mathcal{S}(n_{i2}) + j \cdot p_{\text{clk}}$, respectively, where $j \in \mathbb{Z}_{\geq 0}$. By Equation (8.4), we know that at time $t = \mathcal{S}(n) - 1$, both fanins of n provide their correct values, thus n computes its correct value at time $t = \mathcal{S}(n)$. Moreover, as n_{i1} and n_{i2} produce the same values every p_{clk} phases, the same correct computation also repeats every p_{clk} phases since $t = \mathcal{S}(n)$. Notice that this argument does not require $\mathcal{S}(n_{i1})$ and $\mathcal{S}(n_{i2})$ to be equal. By induction, we conclude that all gates compute and produce the correct value since time corresponding to their assigned levels and every p_{clk} phases afterward.

Finally, by definition of $d(N)$, we know that all combinational outputs o are ready since time $t = \mathcal{S}(o) \leq d(N) \leq \phi_{\text{ro}} + d$, thus at time $t = p_{\text{arch}} \geq \phi_{\text{ro}} + d$, correct values $f_N(\vec{x})$ are presented at the inputs of registers O. Equation (8.6) ensures that register inputs are placed at the correct phase. □

Notice that in this analysis, the requirements for the architectural clock period $p_{\text{arch}} = k \cdot p_{\text{clk}}$ and $p_{\text{arch}} \geq \Phi_{\text{ro}} + d$ must hold regardless of adopting path-balancing or phase-alignment constraints. In other words, the proposed relaxation does not affect architectural clock frequency or latency.

8.2.2 PI Capacity and Phases

Based on the conventional D-latch as shown in Figure 7.3, which adopts the 4-phase clocking scheme, we modify the design in Figure 8.2 to show the possibility for memory devices to

fanins.

have an output capacity larger than 1 and to have their output signal available at multiple phases. In Figure 8.2, buffers are replaced by splitters to drive up to $s_b - 1$ fanouts at various phases, not only phase 4. Adopting such D-latches as registers in a sequential circuit, PIs of the combinational network now have a splitting capacity $s_i = s_b - 1$ (where s_b is usually 3 or 4) instead of 1.

With the modified D-latch design in Figure 8.2, instead of $\Phi_{\text{ro}} = \{4\}$ when adopting D-latches in Figure 7.3, we may use $\Phi_{\text{ro}} = \{3, 4, 5\}$ for a more relaxed phase-alignment requirement because register outputs can be provided at various phases in the feedback loop in D-latch.

8.2.3 Consideration of Clock Skews

The analysis above assumes an ideal clock with zero clock skew. However, in real circuits, clock skews may arise when the clock signal travels along many logic levels. In other words, the activated time of a gate receiving a phase-1 clock closer to the clock source may be earlier than another gate receiving also a phase-1 clock, but further away from the clock source. The difference in the clock timing is called clock skew. One typical superconductor electronics process used to manufacture AQFP circuits is the National Institute of Advanced Industrial Science and Technology (AIST) $10\,\text{kA}\cdot\text{cm}^{-2}$ Nb four-layer high-speed standard process (HSTP). In this process, microstriplines with a ground layer are used to deliver the AC power-clock signals to the AQFPs. A first-order approximation of the transport delay of a $5\,\mu$m long microstripline in this process is approximately $6.20\,\text{ps}\cdot\text{mm}^{-1}$ [Aya+20]. This results in a non-zero clock skew that accumulates along the meandering power-clock network of the AQFPs [Aya+21]. With the existence of a non-zero clock skew, there is an upper limit on how many phases can be skipped without any buffer in between, in addition to the phase alignment constraint.

For large AQFP circuit designs such as a microprocessor, a meandering power-clock network may span across an entire chip which is typically in the range of $5\,\text{mm} \times 5\,\text{mm}$ to $10\,\text{mm} \times 10\,\text{mm}$ in present-day superconductor fabrication processes. The accumulated skew at this scale is significant enough to produce timing errors at GHz-range operating frequencies. In this case, it is important to physically constrain the clock skew by using microwave power dividers [Aya+21] or microwave H-tree networks [He+22] to reduce the physical size of the local meandering microstripline power-clock networks, and thus reduce the accumulated clock skew. Timing characterization of AQFP cells indicate that for 5 GHz sinusoidal clocks, data can still be successfully captured with a clock skew of up to 30 ps between the launching and capturing AQFP [Aya+15; ACY19; Aya+20]. This provides a nominal baseline target for how the power-clock network should be designed, and it also provides an upper limit on how much phase-skipping can be tolerated.

8.3 Impact of Technology Constraints on JJ Count

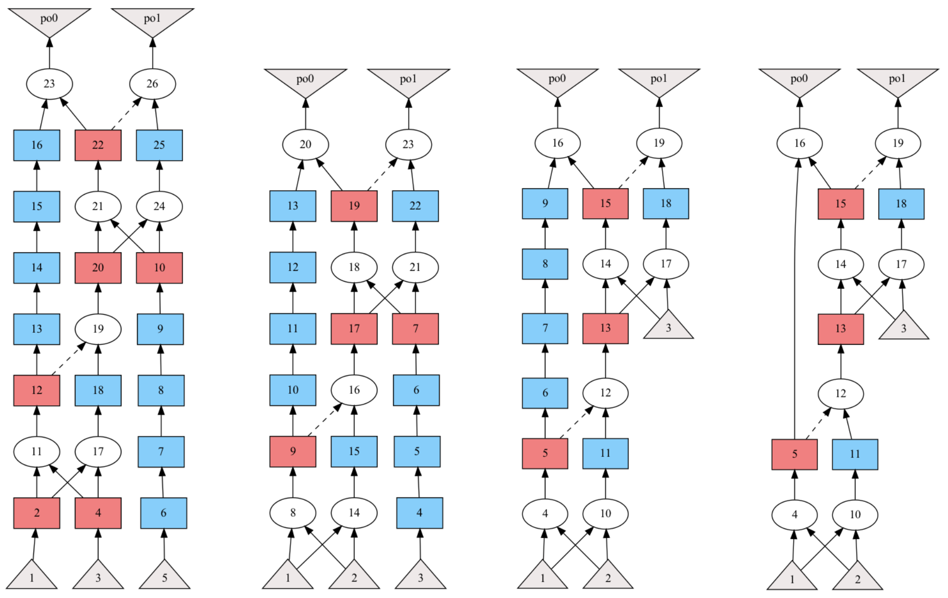
In this section, we demonstrate the impact of the proposed relaxation on technology constraints on the number of buffers, and consequently on the JJ count of an AQFP circuit. To simplify the problem and control unrelated variances, we insert buffers and splitters without modifying the logic structure using an algorithm adapted from the legalization flow described in the next chapter (Chapter 9). After a brief summary in Section 8.3.1 explaining the adaptations made in the algorithm, in Section 8.3.2, a small example circuit is first presented, for which the optimum can be easily derived. Then, in Sections 8.3.3 and 8.3.4, experimental results comparing different constraint formulations are listed.

8.3.1 Buffer/Splitter Insertion Considering Relaxed Constraints

The *AQFP legalization* problem, also called the AQFP *buffer insertion* problem, asks to insert the least buffers and splitters into a logic network, without logic restructuring, to fulfill the technology constraints. This problem will be further described in Chapter 9, including a formal definition, related works, and a proposed flow combining various algorithms. The details are omitted here, but it is worth noting that all existing works on this problem assume the conservative constraints, i.e., path balancing and fanout branching.

To experiment with different formulations of the technology constraints, we adapted the algorithms to support customizable parameters involved in the constraints. These parameters include:

- Buffer's splitting capacity s_b: The maximum out-degree of buffers. This is the same as in previous works.
- PI's splitting capacity s_i: The maximum out-degree of PIs. s_i was fixed to 1 in previous works. However, as discussed in Section 8.2.2, it is possible to have $s_i = s_b - 1$. Thus, we make this an integer parameter to be specified by the user.
- A flag to switch between path balancing (Equations (8.1) to (8.3)) and phase alignment (Equations (8.4) to (8.6)): If phase alignment is adopted, modifications in the algorithms are made. First, levels of PIs and POs are not fixed. Special care is given to ensure that PIs and POs are always assigned to a legal phase with respect to p_{clk} and Φ_{ro}. Finally, chains of single-fanout buffers of a length being a multiple of p_{clk} are removed in a post-processing step.
- Number of phases in a gate-level clock cycle p_{clk}: When adopting path balancing, as in previous works, this parameter is not relevant. However, when relaxing path balancing to phase alignment, p_{clk} is involved in the constraints.
- Possible phase differences between register input and output Φ_{ro}: Set of phases PIs are allowed to be assigned (Equation (8.5)). In previous works, PIs are always assigned to level 0 (Equation (8.2)).



(a) Path-balanced, $s_i = 1$ (16 buffers)
(b) Path-balanced, $s_i = 2$ (13 buffers)
(c) Imbalanced PIs and POs (9 buffers)
(d) Remove buffer chains (5 buffers)

Figure 8.3: Running example of how technology constraints affect the number of buffers in a small circuit.

- If clock skew is of concern, as discussed in Section 8.2.3, then in any unbalanced path, a user-specified maximum phase-skip is ensured.

A possible realistic setting uses phase-alignment constraints and parameters $s_b = 3, s_i = 2, p_{\text{clk}} = 4, \Phi_{\text{ro}} = \{3, 4, 5\}$, which is expected to result in the least number of buffers.

8.3.2 Motivational Example

We use an 1-bit full adder circuit as an example. In Figure 8.3, PIs are at the bottom and POs on top; ellipse nodes are MAJ gates whose constant inputs are neglected for simplicity (i.e., AND gates or OR gates) and negated fanins are dashed; and square blue and red nodes are buffers and splitters. All subfigures show the optimal insertion subject to the specified constraints.

The mapped network when adopting conventional constraints (path balancing and fanout branching, $s_i = 1$) is shown in Figure 8.3 (a), which is the optimal insertion with 16 buffers already shown in state-of-the-art works [LRD22b]. If s_i is increased to 2 as discussed in Section 8.2.2, splitters at the first level are no longer needed, decreasing the network depth by 1 and reducing the number of buffers to 13, as shown in Figure 8.3 (b).

Moreover, as discussed in Section 8.2.1, when enforcing the phase alignment constraint instead of path balancing, the number of buffers further reduces to 5, which is less than a third of the initial mapped network. This adjustment is done in two steps as described in

Table 8.2: Experimental results comparing different constraints.

		Baseline		A		B		A+B		Best				
Register design		$s_i = 1$, $\Phi_{\text{ro}} = \{4\}$		$s_i = 2$, $\Phi_{\text{ro}} = \{3,4,5\}$		$s_i = 1$, $\Phi_{\text{ro}} = \{4\}$		$s_i = 2$, $\Phi_{\text{ro}} = \{3,4,5\}$		$s_i = 2$, $\Phi_{\text{ro}} = \{3,4,5\}$				
Balance PIs & POs		Yes		Yes		No		No		No				
Remove buffer chains		No		No		No		No		Yes				
Bench.	#Gates	#Buf.	#JJs	#Buf.	ΔB	#Buf.	ΔB	#Buf.	ΔB	#Buf.	ΔB	#JJs	ΔJJ	MPS
adder1	7	16	74	13	(19%)	12	(25%)	9	(44%)	5	(69%)	52	(30%)	4
adder8	77	400	1262	341	(15%)	172	(57%)	115	(71%)	87	(78%)	636	(50%)	24
mult8	439	1740	6114	1721	(1%)	1297	(25%)	1305	(25%)	681	(61%)	3996	(35%)	60
counter16	29	80	334	64	(20%)	56	(30%)	52	(35%)	52	(35%)	278	(17%)	20
counter32	82	170	832	158	(7%)	142	(16%)	139	(18%)	131	(23%)	754	(9%)	28
counter64	195	379	1928	360	(5%)	319	(16%)	317	(16%)	309	(18%)	1788	(7%)	36
counter128	428	801	4170	776	(3%)	685	(14%)	680	(15%)	656	(18%)	3880	(7%)	44
c17	6	18	72	5	(72%)	14	(22%)	5	(72%)	5	(72%)	46	(36%)	0
c432	121	904	2534	805	(11%)	582	(36%)	487	(46%)	147	(84%)	1020	(60%)	28
c499	387	1328	4978	1306	(2%)	1299	(2%)	1235	(7%)	407	(69%)	3136	(37%)	24
c880	306	1786	5408	1623	(9%)	982	(45%)	888	(50%)	516	(71%)	2868	(47%)	40
c1355	389	1330	4994	1321	(1%)	1302	(2%)	1242	(7%)	398	(70%)	3130	(37%)	24
c1908	289	1325	4384	1305	(2%)	1181	(11%)	1132	(15%)	364	(73%)	2462	(44%)	28
c2670	368	2036	6280	1812	(11%)	712	(65%)	459	(77%)	351	(83%)	2910	(54%)	32
c3540	794	2339	9442	2226	(5%)	1722	(26%)	1564	(33%)	1060	(55%)	6884	(27%)	44
c5315	1302	6013	19838	5791	(4%)	2743	(54%)	2417	(60%)	1337	(78%)	10486	(47%)	40
c6288	1870	9040	29300	9008	(0%)	5924	(34%)	5886	(35%)	3206	(65%)	17632	(40%)	168
c7552	1394	10243	28850	9521	(7%)	4373	(57%)	4108	(60%)	1860	(82%)	12084	(58%)	56
sorter32	480	544	3968	448	(18%)	544	(0%)	448	(18%)	448	(18%)	3776	(5%)	0
sorter48	880	1008	7296	960	(5%)	1008	(0%)	960	(5%)	960	(5%)	7200	(1%)	0
alu32	1513	14212	37502	13889	(2%)	7976	(44%)	7797	(45%)	1969	(86%)	13016	(65%)	156
Total		55712	179560	53453	(4%)	33045	(41%)	31245	(44%)	14949	(73%)	98034	(45%)	

Section 8.3.1. First, relaxing the constraints on PIs and POs (Equations (8.5) and (8.6) instead of Equations (8.2) and (8.3)) results in Figure 8.3 (c) with 9 buffers. Then, removing buffer chains (Equation (8.4) instead of Equation (8.1)) saves 4 more buffers.

8.3.3 Experimental Results on Constraint Relaxation

Table 8.2 shows the experimental results on the commonly-used benchmark suite consisting of ISCAS benchmarks and some arithmetic circuits[3]. Five sets of constraints are presented and compared. To have a fair comparison, all of them use $s_b = 3$ and $p_{\text{clk}} = 4$ and the mapped networks are obtained using our adapted buffer insertion framework described in Section 8.3.1. Columns "#Bufs." list the number of buffers in the (optimized) mapped networks, columns "#JJs" list the JJ count of the mapped networks (obtained by $\#\text{JJs} = 6 \cdot \#\text{Gates} + 2 \cdot \#\text{Buf.}$), columns "ΔB" and "ΔJJ" list the reduction on buffer count and JJ count, respectively, and column "MPS" list the maximum phase skip.

[3]Available: https://github.com/lsils/SCE-benchmarks

Table 8.3: Experimental results on EPFL benchmarks.

		Baseline		Best		
Register design		$s_i = 1$, $\Phi_{\text{ro}} = \{4\}$		$s_i = 2$, $\Phi_{\text{ro}} = \{3, 4, 5\}$		
Constraint		Path balancing		Phase alignment		
Bench.	#Gates	#Buf.	$R_{\text{B-G}}$	#Buf.	$R_{\text{B-G}}$	ΔB
adder	384	50'046	130.3	1'904	5.0	(96%)
bar	3'016	3'125	1.0	2'310	0.8	(26%)
div	57'300	1'883'971	32.9	148'268	2.6	(92%)
hyp	136'108	9'065'938	66.6	386'735	2.8	(96%)
log2	24'457	129'363	5.3	50'013	2.0	(61%)
max	2'413	71'841	29.8	3'341	1.4	(95%)
multiplier	19'716	103'153	5.2	40'263	2.0	(61%)
sin	4'307	19'261	4.5	8'450	2.0	(56%)
sqrt	23'238	1'796'085	77.3	50'299	2.2	(97%)
square	12'179	90'857	7.5	29'195	2.4	(68%)
arbiter	7'000	28'134	4.0	14'962	2.1	(47%)
cavlc	667	762	1.1	705	1.1	(7%)
ctrl	118	163	1.4	133	1.1	(18%)
dec	304	376	1.2	352	1.2	(6%)
i2c	1'246	2'921	2.3	1'549	1.2	(47%)
int2float	237	321	1.4	260	1.1	(19%)
mem_ctrl	42'714	224'766	5.3	61'114	1.4	(73%)
priority	988	17'546	17.8	1'466	1.5	(92%)
router	267	1'606	6.0	401	1.5	(75%)
voter	7'860	19'619	2.5	15'944	2.0	(19%)
Total/Average		13.5M	20.2	0.8M	1.9	(94%)

Column "Baseline" is the most conservative constraints used in related works [Xu+17; Aya+20; Hua+21; LRD22b; CD23; Fu+23a], i.e., path balancing and fanout branching, plus an additional but realistic constraint that the network depth must be a multiple of $p_{\text{clk}} = 4^4$. Column "A" uses the improved D-latch design discussed in Section 8.2.2, but still adopts path balancing. In contrast, column "B" still uses the classical register design, but does not balance PIs and POs. Column "A+B" combines both improvements. Finally, column "Best" further removes buffer chains in "A+B", shifting from path balancing to phase alignment and achieving the best constraint relaxation proposed in this chapter.

We observe from this experiment that considering phase alignment instead of path balancing reduces about 70% of buffers in AQFP circuits, among which about 40% are balancing PIs and POs, and the other 30% are chains of buffers within the network.

8.3.4 Experimental Results Using Larger Benchmarks

Table 8.3 shows the results of a similar experiment on the EPFL benchmark suite [AGD15], which consists of up to 100× larger benchmarks than in the previous section. For the sake of simplicity, only the settings corresponding to columns "Baseline" and "Best" in Table 8.2 are shown. The number of buffers ("#Buf.") and the buffer-to-gate ratio ("$\mathrm{R}_{\text{B-G}}$", the number of buffers divided by the number of gates) are listed for the two settings, as well as the reduction percentage on buffer count after relaxation ("ΔB").

It can be observed that many benchmarks have a high buffer-to-gate ratio when adopting the conventional conservative constraints, especially the arithmetic circuits (upper half). This is likely due to the imbalanced nature of these circuits. By relaxing the path-balancing constraint to phase alignment, a large portion of path-balancing buffers are eliminated, drastically reducing the number of buffers and making the buffer-to-gate ratio more reasonable. Take the *adder* benchmark as an example, with merely 384 gates in the original network, state-of-the-art buffer insertion algorithms adopting conservative constraints need to insert around 50k buffers to balance every path, 130× of the number of gates. Most JJs in the circuit and energy dissipation are wasted on these buffers. The resulting bulky mapped network also makes the following physical design and fabrication steps difficult. However, simply by relaxing the constraints to phase alignment, only about 1.9k buffers are actually needed, reducing the buffer count by 96%.

8.4 Discussions

8.4.1 Trade-off Between Throughput and Maximum Phase Skip

A disadvantage of replacing path balancing with phase alignment is that the possibility of wave-pipelining is disabled. *Wave-pipelining*, or multi-threaded gate-level pipelining, is a technique to increase throughput by propagating more than one computation in one (architectural) clock cycle, which has been researched for classical CMOS-based digital systems [Bur+98] as well as emerging technologies [Zog+17; Li+22]. One important requirement for a wave-pipelined system is path balancing, thus making AQFP circuits a natural candidate to adopt this technique, although related research has not been proposed yet.

If an AQFP circuit is fully path-balanced, up to $k = p_{\text{arch}}/p_{\text{clk}}$ waves may be propagated between two register stages at the same time, increasing its throughput by $k\times$. When phase alignment is adopted instead to reduce JJ count, a trade-off between throughput and buffer count (thus energy and area) arises. In such case, the number of waves allowed is bounded by the maximum phase skip, or inversely, given a desired throughput, the maximum allowed phase skip must be ensured, which can be achieved with our framework. Related work for the SFQ technology family has been proposed [Li+22], which uses ILP for scheduling and buffer

[4]Many related works do not impose this constraint, although it is necessary. Enforcing this constraint adds about 1.7% buffers on this benchmark suite.

insertion under similar constraints. However, for AQFP, because splitters are also clocked, this formulation cannot guarantee optimality and is also less scalable than our approach. Future AQFP circuit designers may choose path-balanced, wave-pipelined circuits for smaller components requiring higher throughput, and phase-aligned, non-pipelined circuits for larger parts consuming more energy.

8.4.2 n-phase Clocking

Another buffer reduction method leveraging an n-phase clocking scheme has recently been proposed [SAY21]. The basic idea is to multiply the number of phases in one (gate-level) clock cycle by an integer r while keeping the activated period of each gate the same, such that a gate is valid for r phases and any chain of r buffers can be reduced to 1. An example with $p_{\text{clk}} = 4$ and $r = 2$, $n = 8$ is illustrated in Figure 8.4, where the colored areas are the times when gates at the corresponding phase are activated and arrows indicate the transfer of information. In normal 4-phase clocking, information can only be transferred from ϕ_1 to ϕ_2, whereas in 8-phase clocking, information can be transferred from ϕ_1 to ϕ_2 and ϕ_3. Using our terminology, n-phase clocking can be seen as using fractions instead of integers as the range of the schedule, i.e., a gate may be assigned to levels $1/r, 2/r, \ldots$, etc.

The n-phase clocking technique is also very effective in reducing the number of buffers in AQFP circuits, but it does not diminish the value of this work because the relaxation comes from different sources. n-phase clocking relaxes the path balancing constraint by changing the clocking scheme, whereas we develop our argument from analysis of the sequential circuit model. Thus, these two relaxations affect the constraints independently. Instead of comparing against n-phase clocking, we argue that these are two independent techniques that may work in collaboration to achieve the best results and future work remains to formally consider them together. Also, as both techniques have their own drawbacks, engineers may choose between the two depending on the application requirements.

8.4.3 Physical Design and Post-physical-design Legalization

In this chapter, we propose to relax path-balancing constraints to phase alignment, which will have an impact on physical design because current tools generally expect a path-balanced netlist as their input. Although adapting a physical design tool accordingly to generate realistic layouts is beyond the scope of this thesis, Figure 8.3 serves as a good visualization of how a real layout would appear. Moreover, to truly exploit the possible area reduction due to the lower buffer count, the placement algorithm needs to be adapted to allow circuit folding. That is, instead of placing logic gates scheduled at the same level in the same physical row and having as many rows as logic levels, some gates could be placed in different rows with empty slots because of phase skipping. However, this would affect wire lengths and clock synthesis, with additional physical and timing constraints to be carefully considered.

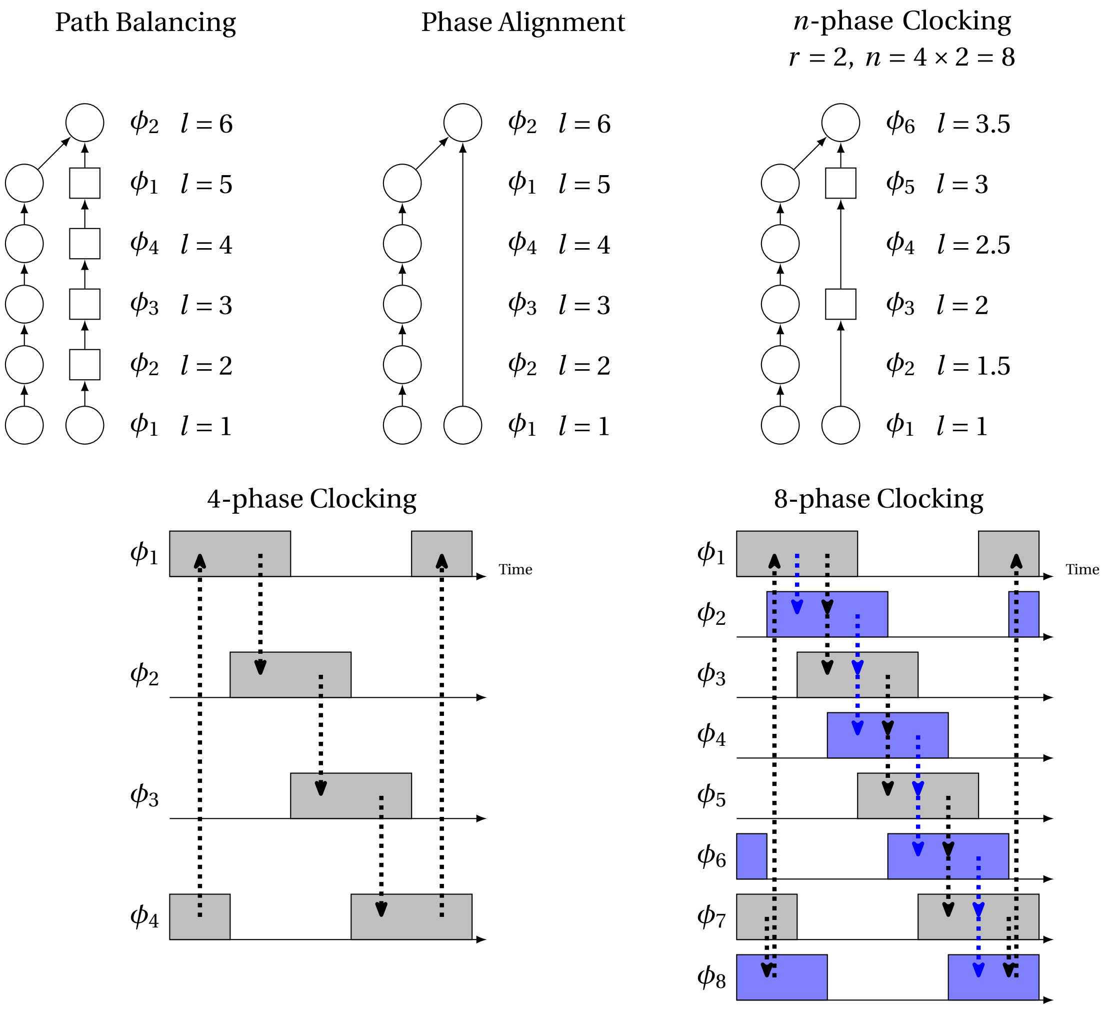


Figure 8.4: n-phase clocking compared to path balancing and phase alignment. ($p_{\mathrm{clk}} = 4$)

The real clock skew between two gates in an AQFP circuit does not only depend on the number of phases in between but also on the microstripline length of the power-clock network between them [Aya+21]. Moreover, interconnect delay of data signals and longer wire lengths must also be considered to ensure the correct operation of an AQFP circuit. If the physical distance between the launching and capturing gates is too long (> 0.7 mm for buffer-to-buffer connections), we may need to insert repeaters or use current boosters. However, these values are only available after physical design and are hard to predict during the buffer insertion stage. Thus, an estimation must be used in buffer insertion. More careful analysis and legalization, which may result in extra buffers being inserted, have to be done during or after physical design. Such overhead may occur in any AQFP synthesis flow regardless of whether adopting the proposals of this work or not, but having a higher phase skip may cause the circuit to be more prone to these issues, especially when operating in high frequency.

Assuming a layout realized similar to Figure 8.3 (d), we expect the power-clock margins to remain unchanged. However, we expect timing margins to reduce because larger phase skipping will likely incur more skew beyond the ideal timing of the capturing clocking phase. Thus, timing-aware placement [Don+22] is important to make sure the circuit still meets sufficient timing margins.

8.5 Summary

In this chapter, we experiment with how assumptions on technology constraints impact AQFP circuit cost and propose possible relaxations. When working with new technologies, formalizing the technology constraints correctly on the chosen abstraction level is important, because if the formulation does not correlate to the underlying technology, the research work that follows becomes meaningless. Indeed, we have shown in this chapter different possibilities in formulating the technology constraints and demonstrated their impact.

As discussed in Section 8.4.1, although relaxing the path-balancing constraint to phase alignment may save a major portion of buffers, such relaxation has the drawback of invalidating wave-pipelining. As a result, this work divides future research on the AQFP buffer insertion problem into two independent directions: On the one hand, considering path balancing makes the problem computationally easier and maintains the possibility of wave-pipelining. On the other hand, considering phase alignment largely reduces JJ count, as shown in Section 8.3, but its optimization problem becomes harder because of the increased flexibility, and wave-pipelining is not applicable anymore.

In the remainder of this thesis, we explore the former direction and adopt the path-balancing constraint without relaxation. The second direction is left for future investigation. We choose to approach the AQFP legalization problem first considering path balancing and fanout branching for the following reasons: 1) This is the “standard” formulation adopted by the community, so it makes our results easier comparable against other works and our implementation easier to be integrated with other tools. 2) With the arguments and analysis presented in this chapter in mind, we develop our algorithms in a parameterized way such that it is easy to switch between different constraint formulations. As path balancing is the stricter constraint, it is easier to apply relaxing optimizations (e.g., removing buffer chains) on a path-balanced mapped network, compared to the other way around.

9 AQFP Technology Legalization by Buffer/Splitter Insertion

9.1 Motivation

One major challenge in AQFP design automation is the legalization of the logic circuit to fulfill two unconventional technology constraints, *path balancing* and *fanout branching*, before physical design. Due to its gate-level clocking property, AQFP gates require all input signals to arrive at the same time, thus buffers have to be inserted on shorter data paths to balance with the longer paths. Moreover, splitters are needed at the output of AQFP gates driving multiple signals, and these splitters are also clocked. Thus, logic circuits generated by technology-independent logic synthesis must be *legalized* for the AQFP technology by inserting buffers and splitters. Legalization of AQFP circuits is essential to unlock its potential of pipelined computation while maintaining correct functionality.

In a legalized AQFP circuit, *buffers and splitters* (B/S) often contribute to over 50% of the JJ count, which is the commonly-used cost metric related to area as well as energy consumption. Thus, optimized algorithms for AQFP legalization are needed to reduce the overhead and increase the scalability of AQFP circuits.

9.2 Problem Formulation

To fulfill the needs in the AQFP technology for fanout-branching and path-balancing, we define the following properties subject to the *splitting capacities* $s_i = 1, s_g = 1$, and $s_b \geq 1$ of PIs, gates, and buffers, respectively.

Definition 9.1. Given a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$,

1. N' is *path-balanced* if there exists a schedule $\mathcal{S}$ of N' such that

$$\forall n_1, n_2 \in V' : (n_1, n_2) \in E' \Rightarrow \mathcal{S}(n_1) = \mathcal{S}(n_2) - 1, \tag{9.1}$$

$$\forall i \in I : \mathcal{S}(i) = 0, \text{ and} \tag{9.2}$$

$$\forall o \in O : \mathcal{S}(o) = d(N'). \tag{9.3}$$

2. N' is *properly-branched* if every PI has an out-degree no larger than $s_i = 1$, every gate has an out-degree no larger than $s_g = 1$, and every buffer has an out-degree no larger than s_b.

3. N' is *legal* if it is both path-balanced and properly-branched.

■

In an AQFP design automation flow, the logic synthesis stage after RTL synthesis and before physical design converts an input specification netlist (represented as, e.g., an *AND-Inverter Graph* (AIG) or a *Majority-Inverter Graph* (MIG)) into a legal mapped network whose gates are all AQFP-compatible. The problem to be solved is formulated as follows:

Problem 1 (AQFP technology mapping)**.** Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in G, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$ such that:

1. N and N' are logically-equivalent.

2. All gates in G' are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation).

3. N' is legal (i.e., path-balanced and properly-branched).

■

Problem 1 may be solved as one problem, or it may be divided into two problems to be solved independently:

Problem 2 (Majority-based logic restructuring)**.** Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in G, find a network $N^* = (V^* = I \cup O \cup G^*, E^*)$, such that:

1. N and N^* are logically-equivalent.

2. All gates in G^* are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation).

■

Problem 3 (AQFP technology legalization)**.** Given a network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and the value of s_b, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$, such that:

1. N' is legal (i.e., path-balanced and properly-branched).
2. $G' = G^*$, and for all gates $g \in G^*$, $\mathrm{FO}(g)$ and $\mathrm{FI}(g)$ remain the same in N' as in N^*.

■

Moreover, for all of the three problems, in addition to finding a network fulfilling the requirements, we also optimize for some common metrics. For the main problem to solve, Problem 1, common optimization objectives are minimizing JJ count (#JJs $= 6\cdot|G'|+2\cdot|B|$) and minimizing JJ depth $d(N')$.

Problem 2 is equivalent to mapping into and optimizing a *Majority-Inverter Graph* (MIG) [AGD16], which is a logic network where all gates are MAJ3 and edges may contain inverters, because AND2 and OR2 gates are equivalent to MAJ3 with a constant (0 and 1, respectively) input. Graph mapping [Tem+22] and MIG optimization [AGD16; Rie+18; LRD21] are well-researched problems with existing algorithms to use. These algorithms usually optimize for MIG size ($|G^*|$) or depth ($d(N^*)$).

In this chapter, we focus on solving Problem 3. Because $G' = G^*$, this problem is often also referred to as the AQFP B/S insertion problem. Minimizing JJ count in Problem 1 is equivalent to minimizing $|B|$ in Problem 3.

9.3 Related Works

(Rapid) Single-Flux Quantum (RSFQ or SFQ) [LS91b] is a sibling superconducting technology of AQFP and has similar path-balancing and fanout-branching constraints, thus also requiring buffer and splitter insertion [KP18; PP18]. However, a key difference between the two technologies makes the problem computationally distinct for them: In SFQ, splitters are not clocked and not considered in path balancing, so fanout branching and path balancing can be considered separately; whereas AQFP splitters are clocked, thus the two constraints must be considered together to discover potential optimizations. The interplay between buffers and splitters makes the B/S optimization problem for AQFP a challenging one.

In the earliest AQFP design automation tools, legalization was done by first inserting splitters (as balanced trees) at the output of all multi-fanout gates, and then inserting buffers on all imbalanced paths [Xu+17]. This was a rather naive approach that guaranteed the correct operation of the AQFP circuit but often resulted in a large portion of JJ count taken by buffers and splitters. Thus, a local optimization technique called retiming [Aya+20] or buffer merging [Cai+19b] was proposed. The basic idea is to move buffers across a multi-fanin gate or a multi-fanout splitter. For example, moving buffers from the fanins of a MAJ3 gate to its fanout reduces buffers by 3× (Figure 8 in [Cai+19b]); alternatively, moving buffers from the fanouts of a splitter to its fanin can be seen as sharing buffers or delayed splitting and also reduces the buffer count (Figure 5 in [Aya+20]). This idea was elaborated in [Cai+19a] as a B/S insertion algorithm using the notion of virtual splitters.

Further improvements on the B/S optimization problem involving more complicated algorithms were made in the following years. In [Hua+21], the authors attempted to localize the optimization problem to a single wire and proposed a locally-optimal algorithm subject to a complex cost function involving maximum and total additional delay and the number of B/S. The local insertion algorithm has a quadratic complexity. In [Fu+23a], the authors proposed to first solve for a schedule of the mapped network, formulated as an ILP problem with a crafted objective function estimating B/S count, followed by another locally-optimal splitter-tree insertion algorithm subject to the same cost function defined in [Hua+21]. This local insertion algorithm has a cubic time complexity.

Exact methods solving for the global size-optimal B/S insertion were also researched. In [LRD22b], the B/S optimization problem was first formulated as a scheduling problem, encoded as an optimization modulo linear integer arithmetic problem, and solved by a *satisfiability modulo theory* (SMT) solver. (This is also described in this chapter in Section 9.4.3.) The global minimum B/S insertion results were obtained for some small benchmarks. Then, an ILP encoding was proposed in [MD23] which led to some improvement in efficiency, and optimal results for some more benchmarks were reported. Whereas size-optimality still remains intractable, depth-optimal B/S insertion has been proved to be solvable in linear time [CD23]. (This is also described in this chapter in Section 9.4.4.)

9.4 Buffer and Splitter Insertion

In this section, we explain how we approach Problem 3. First, in Section 9.4.1, we identify that the AQFP legalization (buffer and splitter insertion) problem is a scheduling problem because once a schedule is given, the minimal-size mapped network can be derived in linear time using an irredundant buffer insertion algorithm (Algorithm 9.1). Thus, various scheduling methods are then discussed, including *as-soon-as-possible* (ASAP) and *as-late-as-possible* (ALAP) scheduling (Section 9.4.2), SMT-based exact scheduling that minimizes buffer count (Section 9.4.3), and depth-optimal scheduling (Section 9.4.4).

9.4.1 Irredundant Buffer Insertion

Claim. The AQFP legalization problem (Problem 3) is a scheduling problem on the unmapped network.

To elaborate on the above claim, we will first introduce the notion of *irredundant* mapped network. Then, we will present Algorithm 9.1 to show how buffers can be inserted irredundantly given a schedule of the unmapped network. Finally, we show in Lemma 9.1 that irredundant networks have the minimum size subject to its schedule.

Definition 9.2. A mapped network is said to be *irredundant* if the following two conditions hold.

1. There is no dangling buffer, i.e., every buffer has at least one outgoing edge.
2. There does not exist any pair of buffers in the same fanout tree, at the same level, and both of them have out-degrees smaller than s_b.

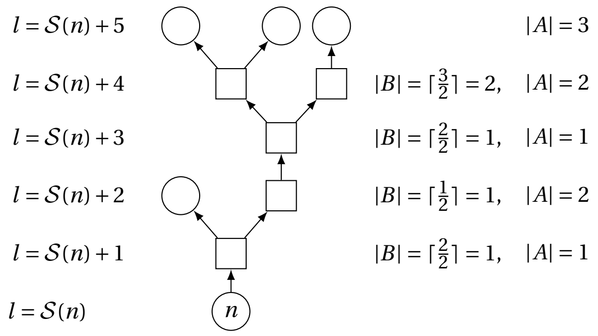
Otherwise, the network is *redundant.* ■

Notice that the local retiming optimization used in [Cai+19a; Aya+20], which pushes buffers from the outputs of a splitter to its input, is subsumed by the definition of irredundant networks. In other words, if a mapped network is irredundant, no optimization can be made with the local retiming technique. This is because local retiming looks for splitters whose fanouts are all buffers and the sum of the fanout counts of these buffers does not exceed the splitting capacity s_b, which violates the second condition in Definition 9.2.

Algorithm 9.1: Irredundant buffer insertion

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and a schedule $\mathcal{S}$ for $N^*$
Output: An irredundant and legal mapped network $N' = (V' = I \cup O \cup G^* \cup B, E')$
1  $N' \leftarrow N^*$
2  foreach $n \in I \cup G^*$ do
3      $l_{\max} \leftarrow \max_{n_o \in \mathrm{FO}(n)} \mathcal{S}(n_o)$
4      $A \leftarrow \{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) = l_{\max}\}$
5      for $l = l_{\max} - 1$ downto $\mathcal{S}(n) + 1$ do
6          Create $\left\lceil \frac{|A|}{s_b} \right\rceil$ buffers at level $l$ in $N'$
7          $B \leftarrow$ the set of newly-created buffers
8          for $i = 1$ to $|A|$ do
9              Remove $n$ from $A[i]$'s fanins in $N'$
10             Add $B[\lceil \frac{i}{s_b} \rceil]$ as $A[i]$'s fanin in $N'$
11         $A \leftarrow B \cup \{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) = l\}$
12     assert $|A| = 1$
13     Add $n$ as $A[1]$'s fanin in $N'$
14 return $N'$
```

For each PI or gate n, Algorithm 9.1 iterates over all levels l between n its fanouts. Initially, the set A contains the fanouts (gates and POs, if any) of n at the highest level $l_{\max}$. At each level l, enough buffers ($|B| = \lceil \frac{|A|}{s_b} \rceil$) are inserted, where $|A|$ is the number of nodes at level $l+1$. Then, n is removed from the fanins of the i-th element in A, and the $\lceil \frac{i}{s_b} \rceil$-th buffer in B is added instead. Finally, A is updated as the newly-created buffers and the fanouts at the current level. Figure 9.1 illustrates an example iteration (of the out-most loop) of Algorithm 9.1, where $s_b = 2$ is assumed.



$l = \mathcal{S}(n) + 5$
$|A| = 3$
$l = \mathcal{S}(n) + 4$
$|B| = \lceil\frac{3}{2}\rceil = 2,\quad |A| = 2$
$l = \mathcal{S}(n) + 3$
$|B| = \lceil\frac{2}{2}\rceil = 1,\quad |A| = 1$
$l = \mathcal{S}(n) + 2$
$|B| = \lceil\frac{1}{2}\rceil = 1,\quad |A| = 2$
$l = \mathcal{S}(n) + 1$
$|B| = \lceil\frac{2}{2}\rceil = 1,\quad |A| = 1$
$l = \mathcal{S}(n)$
n

Figure 9.1: Example sub-network to illustrate Algorithm 9.1. ($s_b = 2$)

Algorithm 9.1 runs in linear time with respect to $\sum_{n\in I\cup G^*} |\mathrm{FO}(n)| \le |E^*|$. It also verifies whether it is possible to build a properly-branched network with the given schedule $\mathcal{S}$. In line 12, the assertion makes sure that the gate or PI n has only one outgoing edge. If this assertion does not hold, then it is impossible to construct a legal mapped network with $\mathcal{S}$ and we say that $\mathcal{S}$ is an illegal schedule. Otherwise, the constructed mapped network is properly branched if the given schedule is legal. It is also path-balanced as each node is connected to a node at exactly one level lower. Moreover, the constructed mapped network is irredundant because in each level in the fanout tree, at most one of the inserted buffer has fanout count smaller than s_b.

With the following lemma, we show that an irredundant network created by Algorithm 9.1 is size-optimal with respect to the given schedule.

Lemma 9.1. *Given an irredundant mapped network $N'_1 = (V'_1 = I \cup O \cup G' \cup B_1, E'_1)$ with a schedule $\mathcal{S}_1$, there does not exist, for the same unmapped network, a smaller mapped network $N'_2 = (V'_2 = I \cup O \cup G' \cup B_2, E'_2)$ with a schedule $\mathcal{S}_2$ such that $\forall n \in I \cup O \cup G', \mathcal{S}_1(n) = \mathcal{S}_2(n)$ and $|B_2| < |B_1|$.*

Proof. Because $|B_2| < |B_1|$, there exists at least one node n whose fanout tree is smaller in N'_2 than in N'_1. Let us denote the two fanout trees as T_1 and T_2, respectively, and the number of buffers at level l in T_1 and T_2 as $b_1(l)$ and $b_2(l)$, respectively. Because the levels of n and its fanouts are the same in the two networks, there exists at least one level l such that

$$b_2(l) < b_1(l). \tag{9.4}$$

Let us consider the highest of such a level so that $b_2(l+1) \ge b_1(l+1)$ and let the number of fanouts of n at level $l+1$ be $o(l+1)$, which is the same in the two networks. The number of edges from level l to level $l+1$ is $b_1(l+1) + o(l+1)$ and $b_2(l+1) + o(l+1)$, respectively, and we have

$$b_2(l+1) + o(l+1) \ge b_1(l+1) + o(l+1). \tag{9.5}$$

Because N'_1 is irredundant, there is at most one buffer at level l in T_1 with out-degree smaller than s_b. In other words, there are at least $s_b \cdot (b_1(l) - 1)$ outgoing edges provided by the other buffers, so

$$b_1(l+1) + o(l+1) > s_b \cdot (b_1(l) - 1). \tag{9.6}$$

On the other hand, in T_2 at level l, $b_2(l)$ buffers can provide at most $s_b \cdot b_2(l)$ outgoing edges, so we have

$$b_2(l+1) + o(l+1) \le s_b \cdot b_2(l). \tag{9.7}$$

Finally, we derive

$$s_b \cdot b_2(l) \ge b_2(l+1) + o(l+1) \ge b_1(l+1) + o(l+1) > s_b \cdot (b_1(l) - 1) \tag{9.8}$$
$$\implies b_2(l) > b_1(l) - 1, \tag{9.9}$$

which is in contradiction to Equation (9.4). Thus, such N_2' does not exist. $\square$

In conclusion, a legal schedule on the unmapped network determines an irredundant and legal mapped network, therefore Problem 3 is equivalent to finding a legal schedule whose corresponding irredundant mapped network is optimal with respect to the given cost metric.

9.4.2 Simple Heuristic Scheduling

To obtain a legal schedule on an unmapped network such that an irredundant legal mapped network can be derived using Algorithm 9.1, we need a scheduling algorithm. As the scheduling problem is well-researched in the context of behavioral-level synthesis [HLH91], we borrow the simplest scheduling algorithms to be used in our problem. The *as-soon-as-possible scheduling* (ASAP) is a greedy algorithm that schedules nodes in a topological order to their lowest possible level according to the schedule of their fanins. In the context of AQFP legalization, to ensure the legality of the schedule, enough levels for a balanced fanout tree are reserved at the output of each multi-fanout node, which is calculated by $\left\lceil \frac{\log(|\mathrm{FO}(g)|)}{\log(s_b)} \right\rceil$.

Another well-known scheduling algorithm is the *as-late-as-possible scheduling* (ALAP), which, conversely, schedules each node to the highest possible level in a reversed topological order. For the upper bound on the maximum levels to schedule the POs, we use $d(N)$ obtained by ASAP.

9.4.3 Exact Scheduling

With the direct relation between a schedule and the corresponding minimal buffer count given by Algorithm 9.1, Problem 3 can be formulated as a *satisfiability modulo theory* (SMT) [Bie+09] problem using linear integer arithmetic as the underlying theory. The primary variables of the instance are integers corresponding to the depth of each gate. Auxiliary variables are used to compute the total number of buffers using Algorithm 9.1. Four types of constraints are encoded:

1. *Bounds:* An upper bound on the network depth is assumed. Lower and upper bounds on the possible levels of each gate can be obtained using ASAP and ALAP, respectively.
2. *Sequencing:* The directed edges are encoded by asserting $\forall g \in G, \forall g_o \in \mathrm{FO}(g) : \mathcal{S}(g) < \mathcal{S}(g_o)$.
3. *Buffer counting:* For each gate or PI n, the number of buffers at the fanout of n is counted by unrolling the for-loop in the following rewritten version of Algorithm 9.1 using relative levels r between n and its fanouts:

```
$|A| \leftarrow |\{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) - \mathcal{S}(n) = r_{\max}\}|$    (9.10)
for $r = r_{\max} - 1$ downto $1$ do
    $|B| \leftarrow \lceil \frac{|A|}{s_b} \rceil$    (9.11)
    $|A| \leftarrow |B| + |\{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) - \mathcal{S}(n) = r\}|$    (9.12)
assert $|A| = 1$    (9.13)
```

   The maximum possible relative level $r_{\max}$ is computed as the difference between the maximum fanout level in the ALAP schedule and the level of n in the ASAP schedule. (9.11) and (9.12) are encoded $r_{\max} - 1$ times using $2 \cdot (r_{\max} - 1)$ auxiliary variables for $|A|$ and $|B|$ in different iterations. Specifically, (9.11) is encoded by the equivalent relation

$$s_b \cdot (|B| - 1) < |A| \leq s_b \cdot |B|, \tag{9.14}$$

   which is a linear relation because s_b is a constant. (9.12) is encoded with the help of the *if-then-else* (ITE) operator to count the number of fanouts at relative level r. Finally, all the auxiliary variables for $|B|$ are summed up as the total buffer count.
4. *Legality:* The legality of $\mathcal{S}$ is ensured by assuming the assertion in (9.13). That is, the last auxiliary variable for $|A|$ should equal to 1.

To find the global minimum, the satisfiability problem is extended to an optimization problem, either by using an optimization modulo theory solver [BPF15] or by imposing an upper bound on the buffer count and iteratively decreasing the bound until the problem becomes UNSAT. The problem has an exponential search space and optimization modulo theory is NP-hard, thus this formulation may be only practical for small networks. Nevertheless, it provides the possibility to understand how good existing and future-developed heuristics are. In [MD23], an ILP encoding based on fanout-bounded synthesis is proposed, which shows some efficiency improvements, and results for more benchmarks are solved. Thus, in Section 9.7.1, we present numbers from [MD23] to compare our heuristics with and omit explicit discussion on the experimental results of our SMT encoding.

9.4.4 Depth-Optimal Scheduling

Disclaimer. This section is based on contributions by collaborator Alessandro Tempia Calvino.[1] This section is an improvement over the simple heuristic scheduling methods in Section 9.4.2, i.e., an algorithm that guarantees depth optimality is proposed, which yields better results. We thus include a summary of the algorithm in this section for completeness reasons, but omit the proofs.

As discussed in Section 9.2, common cost metrics to be considered for AQFP circuits are network size and depth. Unlike in many other technologies where circuit area and delay are often inversely related in a Pareto curve and engineers must trade one for the other, we observe that in the AQFP buffer insertion problem, the size of an irredundant mapped network correlates to the depth of the provided schedule. Intuitively, in Problem 3, the unmapped network and any mapped network have roughly the same number of paths and similar logic sharing (slight differences may only exist in how fanouts are split), and the size of a mapped network is the sum of all path lengths, which is the network depth, minus the sizes of the shared cones. In other words, a larger network depth results in longer (balanced) paths and thus larger network size. Hence, we present scheduling algorithms that also optimize for depth besides being fast (having a linear time complexity) and giving legal results. These algorithms are intended to serve as quick initial scheduling methods that will be further optimized later on (Section 9.5).

Algorithm 9.2: Depth-optimal single node scheduling

```
Input: A node $n$ and a partial schedule $\mathcal{S}$
Output: Level $\mathcal{S}(n)$ assigned to node $n$
1  $l_{prev} \leftarrow \max_{n_o \in \mathrm{FO}(n)} \mathcal{S}(n_o)$
2  $edges \leftarrow 0$
3  foreach $n_o \in \mathrm{FO}(n)$ in a descending order of $\mathcal{S}(n_o)$, let $l = \mathcal{S}(n_o)$ do
4      $splitters \leftarrow \left\lceil \frac{edges}{s_b^{(l_{prev}-l)}} \right\rceil$
5      $edges \leftarrow splitters + 1$
6      $l_{prev} \leftarrow l$
7  while $edges \neq 1$ do
8      $edges \leftarrow \left\lceil \frac{edges}{s_b} \right\rceil$
9      $l_{prev} \leftarrow l_{prev} - 1$
10 $\mathcal{S}(n) \leftarrow l_{prev} - 1$
11 return $\mathcal{S}(n)$
```

Given a partial schedule $\mathcal{S}$ where some nodes, including n but excluding all fanouts of n, have not been assigned a level, Algorithm 9.2 computes the value to be assigned to $\mathcal{S}(n)$, such that the fanout tree of n has the minimum-possible height. This algorithm follows a similar strategy

[1]This chapter is adapted from [Lee+24] (© 2024 IEEE, reprinted with permission) which is a collaboration work with Alessandro Tempia Calvino based on two previous papers [LRD22b; CD23]. Contents in this section and in Section 9.5.2 are summarized from his work [CD23].

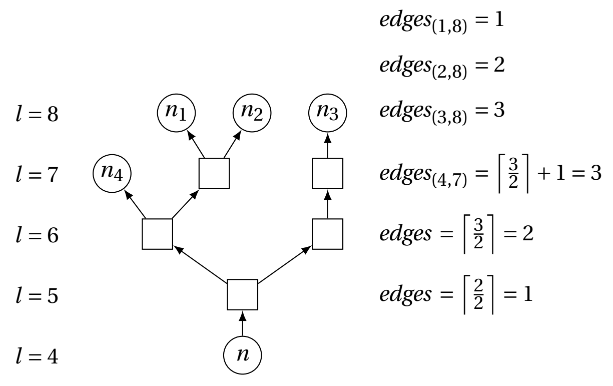

$edges_{(1,8)} = 1$
$edges_{(2,8)} = 2$
$edges_{(3,8)} = 3$
$edges_{(4,7)} = \left\lceil \frac{3}{2} \right\rceil + 1 = 3$
$edges = \left\lceil \frac{3}{2} \right\rceil = 2$
$edges = \left\lceil \frac{2}{2} \right\rceil = 1$
$l = 8$
$l = 7$
$l = 6$
$l = 5$
$l = 4$
n_1
n_2
n_3
n_4
n

Figure 9.2: Example sub-network to illustrate Algorithm 9.2. ($s_b = 2$)

as compared to Algorithm 9.1. Variable *edges* corresponds to $|A|$ in Algorithm 9.1, counting the number of nodes (thus edges) needed to be connected at each level; variable *splitters* corresponds to $|B|$ in Algorithm 9.1, computing the number of splitters (buffers) needed at each level. The foreach-loop (lines 3 to 6) iterates over the fanouts of n in descending order of their levels, and variable l_{prev} keeps the level of the previous iteration. If the level does not change from the previous to the current iteration, variable *splitters* is equal to *edges* because $l_{prev} = l$ and ${s_b}^0 = 1$ (line 4). As a result, *edges* is simply increased by 1 in this iteration, counting the fanout itself (line 5). Otherwise, when a fanout at a lower level is encountered, we compute the minimum number of buffers needed at level l to drive *edges* nodes at level l_{prev} as follows. A complete binary tree of height h has at most 2^h leaves. Similarly, a splitter tree rooted at level l can split into at most ${s_b}^h$ fanouts at level $l + h$. To drive *edges* fanouts at level l_{prev}, at least $\left\lceil \frac{edges}{{s_b}^{(l_{prev}-l)}} \right\rceil$ splitter trees rooted at level l are needed (line 4). Moreover, at most one of them is not full, i.e., they are irredundant. In line 5, this value, plus one for the fanout itself, is used to update variable *edges*. Finally, after all fanouts of n have been processed, the algorithm finds the highest level where *edges* is one to schedule n (lines 7 to 10).

Figure 9.2 shows an example to illustrate Algorithm 9.2, where $edges_{(v,l)}$ indicates the value of variable *edges* when node n_v at level $\mathcal{S}(n_v) = l$ is considered in the foreach-loop (lines 3 to 6). The foreach-loop ends with $l_{prev} = 7$ and *edges* = 3. Then, in the while-loop (lines 7 to 9), *edges* is updated two times before it reaches value 1, resulting in $l_{prev} = 5$. Thus, node n is scheduled at $\mathcal{S}(n) = 4$.

Algorithm 9.2 requires that a node is only scheduled after all of its fanouts have been scheduled. In other words, a reversed topological order is required. Thus, it is suitable to use an ALAP scheduling scheme, which first schedules all POs of a network to an upper bound λ, and then schedules the remaining nodes to the largest-possible level ("as late as possible") in a reversed topological order. We present Algorithm 9.3 for this purpose. It first computes a sufficiently large upper bound λ on the depth of the mapped network for ALAP scheduling, assuming each node would need a balanced splitter tree to drive the maximum fanout in the network. POs are first scheduled at λ. Then, each node is scheduled using Algorithm 9.2 in a reversed

Algorithm 9.3: Depth-optimal ALAP scheduling

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$
Output: A schedule $\mathcal{S}$ for $N^*$ such that its corresponding mapped network $N'$ is legal and $d(N')$ is
        minimal
1  $\lambda \leftarrow d(N^*) \cdot (1 + \max_{n \in V^*} \lceil \frac{\log(|\mathrm{FO}(n)|)}{\log(s_b)} \rceil)$
2  foreach $o \in O$ do
3      $\mathcal{S}_\lambda(o) \leftarrow \lambda$
4  foreach $n \in I \cup G^*$ in a reversed topological order do
5      $\mathcal{S}_\lambda(n) \leftarrow$ schedule_node($n, \mathcal{S}_\lambda$)                // alg. 9.2
6  $l_{\min} \leftarrow \min_{i \in I} \mathcal{S}_\lambda(i)$
7  foreach $i \in I$ do
8      $\mathcal{S}(i) \leftarrow 0$
9  foreach $n \in O \cup G^*$ do
10     $\mathcal{S}(n) \leftarrow \mathcal{S}_\lambda(n) - l_{\min}$
11 return $\mathcal{S}$
```

topological order. Finally, to obtain a schedule independent of the value of λ, post-scheduling correction is applied: PIs are moved to level 0 to fulfill Equation (9.2), and the levels of all other nodes are reduced by the smallest PI level before correction. This algorithm has a linear time complexity with respect to the network size.

In conclusion, Algorithm 9.3 finds a schedule for an unmapped network in linear time. Followed by Algorithm 9.1, a mapped network is obtained, which is guaranteed to be legal and depth-optimal. We omit the formal proofs in this thesis, but they can be found in [Lee+24]. Starting from such an initial schedule (and its corresponding mapped network), we can further optimize it for size using the heuristic optimization algorithms to be presented in the next section (Section 9.5). As heuristics are often biased by the starting point, having more than one different initial schedules may be beneficial. Thus, in addition to the ALAP-based scheduling (Algorithm 9.3), an ASAP-based depth-optimal scheduling algorithm is also proposed. We omit the details in this thesis, but they can be found in [Lee+24].

9.5 Buffer and Splitter Optimization

The scheduling-based legalization approach presented in the previous section allows us to find one (or two) legal mapped network that is (are) depth-optimal. In some scenarios, this may already be good enough, but it is still possible to further optimize the obtained mapped network to reduce its size. In this section, given a mapped network, we attempt to find a better schedule to minimize $|B|$. Two orthogonal heuristic algorithms are proposed in Sections 9.5.1 and 9.5.2, and then combined as a portfolio flow in Section 9.5.3.

Algorithm 9.4: Chunk construction

```
Input: An initial gate $g_0$
Output: A chunk $C$ and its interfaces $T$
1  $C \leftarrow \{g_0\}$
2  $Q \leftarrow \{(g_0, g) : g \in \mathrm{FI}(g_0) \cup \mathrm{FO}(g_0)\}$
3  $T \leftarrow \emptyset$
4  while $Q \neq \emptyset$ do
5      $(g_c, g_e) \leftarrow \mathrm{pop}(Q)$
6      if $g_e \in C$ then continue
7      if $g_c$ and $g_e$ are close then
8          $C \leftarrow C \cup g_e$
9          $Q \leftarrow Q \cup \{(g_e, g) : g \in \mathrm{FI}(g_e) \cup \mathrm{FO}(g_e)\}$
10     else
11         $T \leftarrow T \cup \{(g_c, g_e)\}$
12 return $C, T$
```

9.5.1 Chunked Movement

The *chunked movement* technique attempts to move groups of nodes up or down to reduce the total number of buffers. *Moving* a gate g up (down) by l levels means that $\mathcal{S}(g)$ is increased (resp. decreased) by l while the levels of the other gates remain the same. During the process, we always ensure that the network is legal and buffers are inserted irredundantly using Algorithm 9.1. A movement is *legal* if the network remains legal after the movement. For example, if a gate g has a fanout g_o at level $\mathcal{S}(g_o) = \mathcal{S}(g) + 1$, then moving g up alone is not legal. Similarly, if a gate g has more than one fanout, then moving any of its fanouts down to level $\mathcal{S}(g) + 1$ is not legal because there must be a splitter occupying the only outgoing edge of g at $\mathcal{S}(g) + 1$. We observe that sometimes it is impossible to legally move a single gate, or that moving it alone does not reduce the total buffer count. However, rearranging some neighboring gates together might eventually lead to further reduction. Thus, we propose to identify groups of connected gates and move them together as *chunks*, defined as follows.

A gate g and one of its fanouts $g_o \in \mathrm{FO}(g)$ are said to be *close* if either one of the following conditions holds:

1. $|\mathrm{FO}(g)| = 1$ and $\mathcal{S}(g_o) = \mathcal{S}(g) + 1$.
2. $|\mathrm{FO}(g)| > 1$ and $\mathcal{S}(g_o) = \mathcal{S}(g) + 2$.

If a gate g and its fanout g_o are not close, then there is flexibility at the output of g and the input of g_o. A *chunk* is a set C of closely-connected gates. Seen as a group together, it has multiple incoming and outgoing edges, called the *input interfaces* (IIs) and *output interfaces* (OIs), respectively. An interface is an ordered pair (g_c, g_e) of a gate in the chunk $g_c \in C$ and an external gate $g_e \notin C$, and either $g_e \in \mathrm{FI}(g_c)$ (for an II) or $g_e \in \mathrm{FO}(g_c)$ (for an OI).

Algorithm 9.4 illustrates how a chunk is identified. Starting from an initial gate g_0, a chunk is

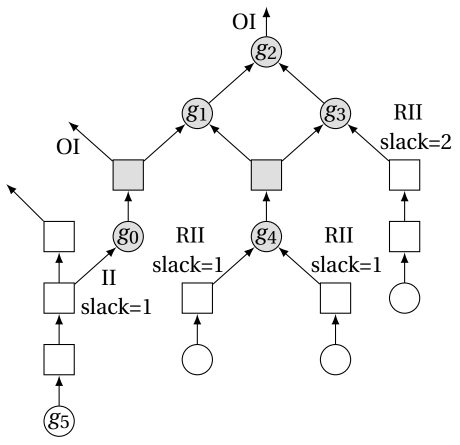


Figure 9.3: Example sub-network showing a chunk (in grey).

formed by exploring its fanins and fanouts and adding gates into the chunk if they are close (line 8), or recording an input or output interface otherwise (line 11). When a new gate is added to the chunk, its fanins and fanouts are also explored (line 9). The queue Q stores the edges to be checked next.

By definition, a chunk has flexibilities at all of its interfaces. Moreover, the set of all chunks in a mapped network forms a partitioning of all gates. Figure 9.3 shows an example chunk. Starting from the initial gate g_0, closely-connected gates g_1, g_2, g_3, g_4 are added into the chunk in the respective order. The gate g_1, for example, cannot be moved up nor down legally without moving other gates at the same time. Also, although the gate g_0 can be legally moved down, moving it alone would only incur more buffers. However, if the entire chunk is moved down together by one level, one buffer is saved, which is analyzed as follows.

To see how many levels a chunk can be moved, a *slack* is computed at each interface. For an input interface (g_c, g_e),

$$\text{slack}(g_c, g_e) = \begin{cases} \mathcal{S}(g_c) - \mathcal{S}(g_e) - 1, \text{ if } |\text{FO}(g_e)| = 1 \\ \mathcal{S}(g_c) - \mathcal{S}(g_e) - 2, \text{ otherwise.} \end{cases} \tag{9.15}$$

For an output interface, g_c and g_e are exchanged in Equation (9.15). When trying to move a chunk down, the maximum number of levels we can move is the minimum slack at all input interfaces; when moving a chunk up, it is the minimum slack at all output interfaces.

We further classify input interfaces as relevant or not. An input interface (g_c, g_e) is said to be a *relevant input interface* (RII) if

$$\forall g_o \in \text{FO}(g_e), g_o \notin C : \mathcal{S}(g_o) > \mathcal{S}(g_c). \tag{9.16}$$

For example, in Figure 9.3, (g_0, g_5) is not an RII because g_5 has another fanout at a higher level than $\mathcal{S}(g_0)$, so when g_0 is moved, no buffer is added or eliminated at this interface.

We decide to move a chunk up or down on whether there are more OIs or RIIs. If a chunk has x OIs and y RIIs, moving the chunk up by l levels eliminates $l \cdot (x - y)$ buffers (if $x > y$), and moving the chunk down eliminates $l \cdot (y - x)$ buffers (if $y > x$). In Figure 9.3, there are 3 RIIs and 2 OIs, and the minimum slack at all IIs is 1, thus moving the chunk down by 1 level reduces 1 buffer.

Overall, the chunked movement algorithm iteratively constructs a chunk using Algorithm 9.4 for each node that is not yet in a chunk and tries to move the chunk up or down, applying the movement only when it is legal and beneficial.

9.5.2 Retiming

Disclaimer. This section is based on contributions by collaborator Alessandro Tempia Calvino.[1] This section illustrates a buffer and splitter optimization algorithm orthogonal to the chunked movement method in Section 9.5.1, which, in combination, yields better results. We thus include a summary of the algorithm in this section for completeness reasons, but omit some details.

The optimization of buffers and splitters in an AQFP circuit is reminiscent of the register minimization problem called retiming. *Minimum register retiming* is the problem of relocating the registers of a circuit in order to minimize their number while preserving the functionality. Retiming is formulated as a linear problem dual to the minimum-cost flow problem for which many polynomial algorithms exist [LS91a]. In this section, we propose the AQFP B/S retiming algorithm, which minimizes buffers and splitters in an AQFP network, similar to how registers are minimized in minimum register retiming. Previous work applied a retiming-like optimization to AQFP logic [Aya+20; Cai+19a]. However, their approach does not perform global retiming but moves buffers locally from the output of splitters to the input. This optimization is subsumed by Algorithm 9.1 in the definition of irredundant mapped networks.

Minimizing the number of buffers can be seen as maximizing sharing of buffers on multiple paths. Without accounting for fanout-branching, e.g., assuming that buffers have an infinite splitting capability, the minimum number of buffers is achievable in polynomial time using a minimum register retiming algorithm considering each buffer as a register. Retiming preserves the path-balancing constraint since each path traverses the same number of registers before and after retiming. As mentioned in Section 9.3, previous works successfully applied this idea to the RSFQ technology family [KP18], but when the fanout-branching constraint in AQFP comes into consideration, splitter relocation is conditional on respecting the splitting capacity. Hence, retiming is only a heuristic for minimizing the buffer count instead of an optimal algorithm.

Figure 9.4 (a) shows an example mapped sub-network under retiming, where $s_b = 3$ is assumed.

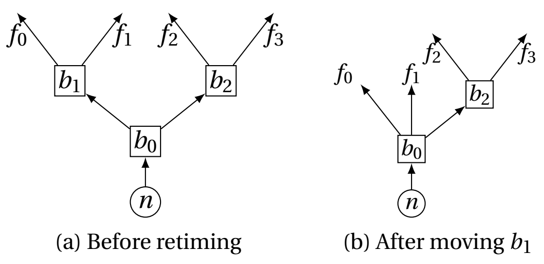
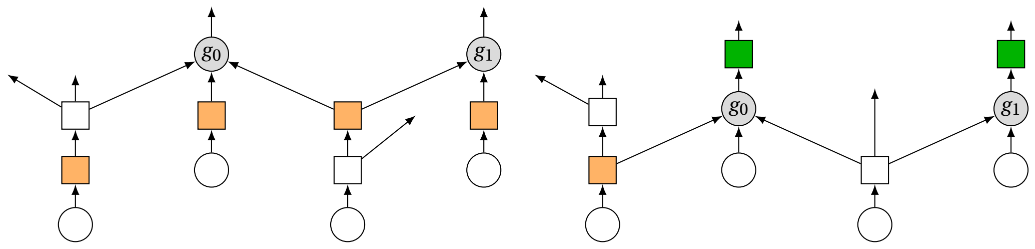


Figure 9.4: Example sub-network for retiming. ($s_b = 3$)

This sub-network is redundant because b_1 and b_2 have out-degree $2 < s_b$ (Definition 9.2). Indeed, a mapped network can become redundant temporarily during retiming. Not all buffers can be retimed at the same time, and this example shows two such cases. First, b_0 cannot be retimed because its movement would increase the fanout count of n to 2, violating the fanout constraint of gates ($s_g = 1$). Second, only one of the splitters b_1 and b_2 can be selected for retiming since the movement of both of them would increase the fanout count of b_0 to 4, violating the fanout constraint of buffers ($s_b = 3$). Also, fanouts of splitters in the same fanout tree originating from the same gate are exchangeable, and such exchanges may affect possible retiming optimizations. For example, instead of $\text{FO}(b_1) = \{f_0, f_1\}$, $\text{FO}(b_2) = \{f_2, f_3\}$ in Figure 9.4 (a), $\text{FO}(b_1) = \{f_0, f_2\}$, $\text{FO}(b_2) = \{f_1, f_3\}$ is also possible and may unlock more retiming on b_1 and b_2. Figure 9.4 (b) shows the fanout tree after the relocation of splitter b_1 to its transitive fanout cone (not shown).

The retiming problem is formulated as a binary maximum-flow problem similar to [HMB07], which separates flow computation into forward and backward directions. The algorithm performs an optimization loop in each direction until no more improvements can be made. A loop starts by selecting a set of buffers, which can be relocated without exceeding the splitting capacity of their fanin nodes, to be retimed. In the case of mutually exclusive selections (i.e., two splitters cannot be retimed at the same time), one is picked randomly. Each selected buffer is a source and a sink of a unitary flow. Then, the algorithm computes the binary maximum flow using the augmenting path algorithm to obtain the minimum cut. If there is a reduction in buffer count, the selected buffers are moved to the new position. Since retiming movements may create redundant fanout trees, at the end of the algorithm, fanout trees are reconstructed irredundantly using Algorithm 9.1.

An example of a forward retiming iteration is depicted in Figure 9.5, where $s_b = 3$ is assumed. The algorithm selects four buffers in the initial sub-network (Figure 9.5 (a), orange) to be retimed. After retiming (Figure 9.5 (b)), three of the selected buffers are removed and two new buffers (green) are inserted. The number of buffers is reduced from 6 to 5 while maintaining the same path lengths.



(a) Initial sub-network (b) Optimized sub-network

Figure 9.5: Example of forward retiming. ($s_b = 3$)

Algorithm 9.5: Buffer and splitter optimization

Input: Mapped network N'_{init}
Output: Optimized mapped network N'_{opt}

```
1 $N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{init}}$)                // section 9.5.2
2 repeat
3     $N'_{\text{opt}} \leftarrow N'_{\text{tmp}}$
4     $N'_{\text{tmp}} \leftarrow$ chunked_movement($N'_{\text{opt}}$)    // section 9.5.1
5     $N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{tmp}}$)         // section 9.5.2
6     $N'_{\text{tmp}} \leftarrow$ randomize($N'_{\text{tmp}}$)
7 until $|N'_{tmp}| \geq |N'_{opt}|$
8 return $N'_{opt}$
```

9.5.3 Buffer and Splitter Optimization Flow

Algorithm 9.5 describes our optimization flow. It combines chunked movement and retiming to achieve better results than the individual algorithms. Additionally, we use a randomization function to pick different random fanout groupings when constructing splitter trees to change the structure of the circuit and unlock further optimizations.

9.6 Technology Legalization Flow

In Section 9.4, we presented algorithms to obtain an initial scheduling (Section 9.4.4) and to insert buffers irredundantly (Section 9.4.1). In Section 9.5, we presented optimization algorithms to further reduce the buffer count of a mapped network. Combining everything together, a technology legalization flow is presented in Algorithm 9.6. Two initial scheduling, ALAP and ASAP are obtained with the depth-optimal scheduling algorithms and result in two mapped networks by inserting buffers irredundantly. Then, the two mapped networks are optimized independently using the portfolio optimization flow. Finally, the better one with a smaller size is adopted.

Algorithm 9.6: AQFP technology legalization flow (solves Problem 3)

```
Input: MIG network $N^*$
Output: Mapped network $N'$
1  $\mathcal{S}_{\text{ALAP}} \leftarrow \text{ALAP}(N^*)$                                   // alg. 9.3
2  $\mathcal{S}_{\text{ASAP}} \leftarrow \text{ASAP}(N^*, \mathcal{S}_{\text{ALAP}})$
3  $N'_{\text{ALAP}} \leftarrow \text{insert\_buffers}(N^*, \mathcal{S}_{\text{ALAP}})$      // alg. 9.1
4  $N'_{\text{ASAP}} \leftarrow \text{insert\_buffers}(N^*, \mathcal{S}_{\text{ASAP}})$      // alg. 9.1
5  $N'_{\text{ALAP}} \leftarrow \text{optimize}(N'_{\text{ALAP}})$                          // alg. 9.5
6  $N'_{\text{ASAP}} \leftarrow \text{optimize}(N'_{\text{ASAP}})$                          // alg. 9.5
7  if $|N'_{ALAP}| < |N'_{ASAP}|$ then
8      return $N'_{ALAP}$
9  else
10     return $N'_{ASAP}$
```

9.7 Experimental Results

In this section, we present experimental results of our methods solving Problem 3 alone (Section 9.7.1). We also demonstrate in Section 9.7.2 the scalability of the proposed AQFP legalization algorithm using much bigger benchmarks. To be consistent with previous works that we compare to, we use $s_b = 4$ for the splitting capacity of buffers.

9.7.1 Technology Legalization and Buffer Optimization

First, we compare the performance of our AQFP legalization and optimization flow (Algorithm 9.6) against the state-of-the-art (SoTA) on solving the same problem [Fu+23a]. For the sake of completeness, we list all of the benchmarks used in the first work on AQFP B/S insertion [Cai+19a] in Table 9.1, but the totals are computed only with the benchmarks presented in [Fu+23a]. The number of gates ($|G^*|$) and the depth ($d(N^*)$) of the initial MIGs, as well as the number of buffers ($|B|$), the JJ count (#JJs) and the depth ($d(N')$) of the mapped networks are listed. Moreover, the runtime (Time) used by our flow is presented. Unfortunately, the runtime data was not presented in [Fu+23a]. In the last column, we list the known global optimum results obtained by ILP solving [MD23] to have an idea of how far the heuristics are from optimal. Some of the numbers are only an upper bound because the ILP formulation could not be solved within a reasonable runtime, and some of the benchmarks are too big for the ILP solver to return any partial result.

From Table 9.1, we can see that the heuristic methods achieve optimum for the smaller benchmarks and are fairly close to optimum for most of the benchmarks. While our flow obtains slightly worse results in average size than SoTA, the difference is very small (0.96% in number of buffers and 0.5% in JJ count). Thanks to the depth-optimal scheduling, we obtain a better depth in one benchmark (c7552). Most importantly, these results are obtained using short runtime. Thus, our flow can be used in design space exploration, where legalization is called extensively, such that large improvements can be achieved (Section 10.5).

Table 9.1: Technology legalization results comparing to the state-of-the-art and global optimum.

	MIG N^*		SoTA [Fu+23a]			Ours (Algorithm 9.6)				Global optimum [MD23]		
Bench.	$\|G^*\|$	$d(N^*)$	$\|B\|$	#JJs	$d(N')$	$\|B\|$	#JJs	$d(N')$	Time (s)	$\|B\|$	#JJs	$d(N')$
adder1	7	4	-	-	-	16	74	8	0.00	16	74	8
adder8	77	17	-	-	-	371	1204	33	0.01	371	1204	33
mult8	439	35	1681	5996	70	1690	6014	70	0.18	≤1724	≤6082	≤70
counter16	29	9	66	306	17	65	304	17	0.00	65	304	17
counter32	82	13	156	804	23	154	800	23	0.01	154	800	23
counter64	195	17	351	1872	30	347	1864	30	0.02	347	1864	30
counter128	428	22	755	4078	38	747	4062	38	0.07	747	4062	38
c17	6	3	-	-	-	12	60	5	0.00	12	60	5
c432	121	26	829	2384	37	839	2404	37	0.02	829	2384	37
c499	387	18	1173	4668	29	1173	4668	29	0.09	1173	4668	29
c880	306	27	1536	4908	40	1511	4858	40	0.15	-	-	-
c1355	389	18	1186	4706	29	1184	4702	29	0.06	1178	4690	29
c1908	289	21	1253	4240	34	1234	4202	34	0.09	1232	4198	34
c2670	368	21	1869	5954	28	1912	6032	28	0.32	≤1804	≤5816	≤28
c3540	794	32	1963	8690	52	1943	8650	52	0.81	≤1926	≤8516	≤52
c5315	1302	26	5505	18942	40	5640	19092	40	2.06	≤6260	≤20332	≤42
c6288	1870	89	8832	28884	179	8647	28514	179	2.56	-	-	-
c7552	1394	33	6768	21908	58	7437	23238	56	4.20	-	-	-
sorter32	480	15	-	-	-	480	3840	30	0.06	480	3840	30
sorter48	880	20	-	-	-	880	7040	35	0.20	880	7040	35
alu32	1513	100	13976	37030	169	13836	36750	169	2.74	-	-	-
Total[1]			47899	155370	873	48359	156154	871	13.38			

[1]Excluding benchmarks missing in SoTA.

9.7.2 Scalable AQFP Legalization

To demonstrate the scalability of our AQFP legalization approach, we use the largest 10 benchmarks in the EPFL benchmark suite [AGD15] for experiment, which are 10×-100× in size compared to the benchmarks generally used in previous works on AQFP logic synthesis. The MIGs are obtained using delay-oriented graph mapping [Tem+22]. In Table 9.2, we compare our results obtained using a simple depth-optimal legalization flow (Algorithm 9.3 followed by Algorithm 9.1, column "D.-opt. legal.") as well as depth-optimal legalization with further optimization (Algorithm 9.6, column "D.-opt. legal.+opt.") against results of non-depth-optimal legalization with optimization presented in [LRD22b] (column "Non.-d.-opt. legal.+opt."). A timeout limit of 300 seconds is enforced. From this experiment, we can see that simple legalization without optimization is very fast, so such a flow can still be used in design space exploration even when benchmarks are large. Comparing the mapped network depths, the proposed depth-optimal scheduling reduces the depth by about 9% on average.

Table 9.2: Technology legalization results on the largest EPFL benchmarks

	MIG N^*		Non-d.-opt. legal.+opt. [LRD22b]			D.-opt. legal. (alg. 9.3 + alg. 9.1)			D.-opt. legal.+opt. (alg. 9.5)		
Bench.	$\|G^*\|$	$d(N^*)$	$\|B\|$	$d(N')$	Time (s)	$\|B\|$	$d(N')$	Time (s)	$\|B\|$	$d(N')$	Time (s)
div	57300	2217	2084772	4918	271.71	1881255	4371	0.87	-	4371	>300
hyp	136109	8762	-	17910	>300	9035578	17246	2.78	-	17246	>300
log2	24456	200	98047	414	194.92	129547	379	0.10	86705	379	64.18
multiplier	19710	133	79651	286	13.21	102005	264	0.08	63414	264	43.50
sin	4303	110	17470	225	5.67	18905	188	0.01	14886	188	4.12
sqrt	23238	3366	1751742	8191	5.64	1791005	6628	0.49	1343705	6628	284.10
square	12180	126	60552	256	42.71	89516	251	0.03	63630	251	18.30
arbiter	7000	59	31011	65	5.80	27566	63	0.01	25721	63	1.28
mem_ctrl	42758	73	305689	182	87.86	216927	114	0.27	215202	114	10.55
voter	7860	47	18044	99	5.43	19263	86	0.01	15736	86	0.92

9.8 Summary

In this chapter, we first establish that the AQFP legalization problem is a scheduling problem and propose two depth-optimal scheduling algorithms. Then, the obtained schedules may be further optimized for size using the proposed chunked movement and retiming techniques. Experimental results show that our legalization flow obtains similar, near-optimal quality as the state-of-the-art ILP-based algorithm within very little runtime. Moreover, our approach is flexible in runtime budget as the optimization part can be skipped. As both irredundant buffer insertion and depth-optimal scheduling have linear time complexity, scalability is guaranteed. We demonstrate legalization results on benchmarks 10× to 100× larger than what any other related works could handle.

10 AQFP Logic Synthesis Toolbox

As discussed in Section 9.2, the AQFP technology mapping problem (Problem 1) can be divided into two sub-problems, MIG restructuring (Problem 2) and AQFP legalization (Problem 3). Solving the two sub-problems together leads to a high problem complexity and has to rely on a pre-computed database that is locally optimal [MRM21]. Hence, we propose to solve the two sub-problems untangled, but mixed and interleaved in multiple iterations to enhance QoR. It is essential for the algorithms used to solve both sub-problems to be efficient, such that more iterations can be done in a reasonable runtime and achieve better results.

In this chapter, we first review related works on the broader problem of AQFP technology mapping in Section 10.1 and summarize MAJ-based logic synthesis algorithms in Section 10.2, including existing works in the literature and a high-effort MIG resubstitution algorithm combining elements proposed in this thesis. Then, in Section 10.3, we present our AQFP technology mapping solution combining MIG optimization (Section 10.2) and AQFP technology legalization (Chapter 9) with the on-the-fly design space exploration methodology introduced in Chapter 5. We also discuss verification for AQFP synthesis in Section 10.4. Finally, we present experimental results on the AQFP technology mapping problem, utilizing all mentioned elements in our AQFP logic synthesis toolbox. The relationship between each section in this chapter and various chapters in this thesis is outlined in Figure 10.1.

10.1 Related Works

Existing AQFP logic synthesis flows can be categorized into two approaches: solving Problem 2 and Problem 3 separately, or considering Problems 2 and 3 together. The earliest works took the first approach to adapt available CMOS-based design automation tools for AQFP [Xu+17; Aya+20]. Problem 2 was addressed by AND-based technology-independent logic synthesis followed by technology mapping into an AQFP-compatible library, and Problem 3 was solved separately in an additional buffer insertion stage before physical design. Later, to better leverage the intrinsic MAJ function in AQFP circuits, MAJ-based logic synthesis was adopted [Cai+19b; Tes+21]. At this time, Problem 3 was still solved separately using the naive

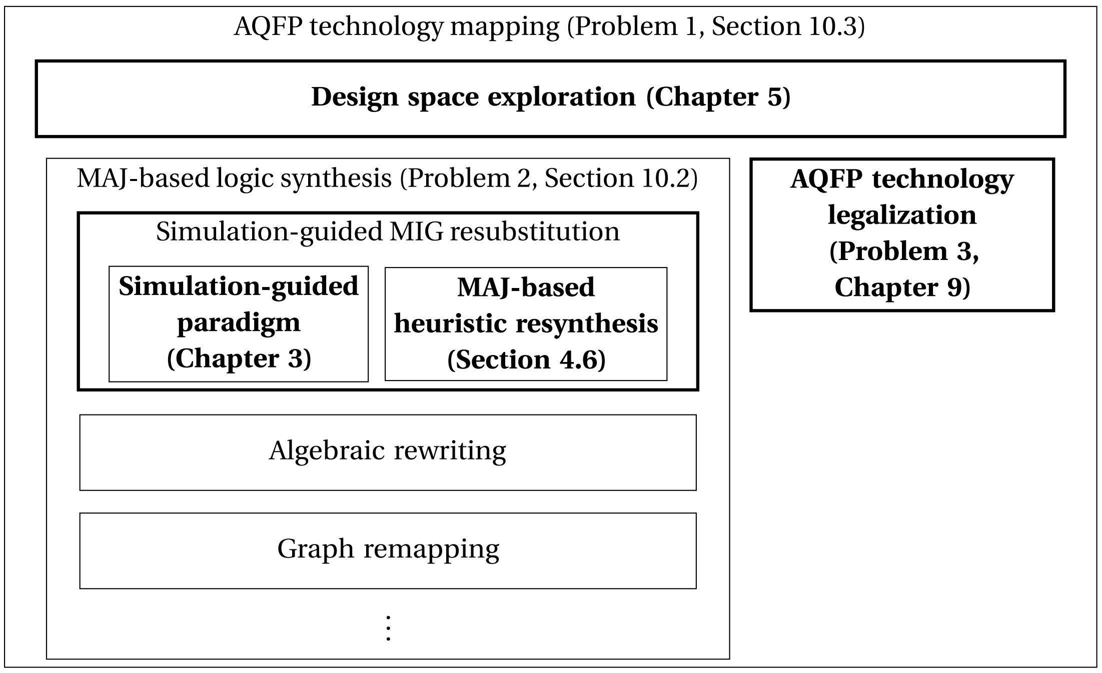

AQFP technology mapping (Problem 1, Section 10.3)
Design space exploration (Chapter 5)
MAJ-based logic synthesis (Problem 2, Section 10.2)
Simulation-guided MIG resubstitution
Simulation-guided paradigm (Chapter 3)
MAJ-based heuristic resynthesis (Section 4.6)
Algebraic rewriting
Graph remapping
⋮
AQFP technology legalization (Problem 3, Chapter 9)

Figure 10.1: Integration of algorithms in various chapters as an AQFP synthesis flow.

insertion approach introduced in Section 9.3.

Although solving the two problems separately is easier, it is hard to predict the impact of legalization in the logic restructuring stage. The smallest MIG in size may not be still the smallest after legalization. Thus, in [MRM21], the authors proposed to consider the two problems together and optimize directly for the final cost function. A database of optimal AQFP sub-circuits is used in restructuring, and legalization is done during the process. This algorithm was used in a flow consisting of graph mapping, AQFP resynthesis, and post-synthesis buffer optimization [Meu+22].

The latest work on AQFP synthesis, presenting currently the best results, took the first approach (separating the two problems) and used Bayesian optimization to find the best MIG restructuring script with respect to the actual AQFP cost after legalization [Fu+23b].

10.2 MAJ-Based Logic Synthesis

MIG was proposed as an alternative technology-independent logic representation with an advantage in depth optimization especially in arithmetic circuits [AGD16]. Due to the special properties of some emerging technologies including AQFP, MIG also become a good logic synthesis data structure for these technologies [Tes+21]. Various logic synthesis and optimization algorithms have been proposed and tailored for MIGs.

To convert an AIG into an MIG, the simplest way is to translate each AND2 gate into an MAJ3 gate with a constant 0 input. Alternatively, a versatile graph mapping algorithm can also map from AIGs (or other types of networks) to MIGs while optimizing for depth and/or size in the process [Tem+22]. The graph mapping algorithm can also be used to optimize MIGs by remapping (i.e., mapping from an MIG into an MIG), in which case it is similar to a cut rewriting algorithm.

Many common logic optimization algorithms originally developed for AIGs can also be applied to MIGs with little adaptation, such as cut rewriting, functional reduction, and balancing. Tailored MIG optimization algorithms include algebraic rewriting, which applies special Boolean algebraic rules to reduce MIG depth [AGD16], and (enumeration-based) resubstitution, which resynthesizes a small part of the network using majority gates to reduce MIG size [Rie+18].

Whereas the above algorithms are relatively lightweight with faster runtimes and limited QoR, in pursuit of additional quality improvements when other algorithms saturate, we apply high-effort simulation-guided MIG resubstitution. By adopting the simulation-guided paradigm (Chapter 3, Section 3.5), the window size is unlimited and more divisors can be considered. Moreover, global satisfiability don't cares are naturally considered when using the simulation signatures computed by global simulations. By leveraging the heuristic resynthesis algorithm (Chapter 4, Section 4.6), larger dependency circuits may be found for roots with a large enough MFFC size. This extends the search space for optimization candidates and creates more gain.

To diversify the set of MIG optimization scripts for better results in design space exploration, we also form complex flows consisting of converting the MIG back into an AIG, applying an AIG-based optimization flow, and then mapping back into MIG. As discussed in Section 5.4, such a decompressing-compressing strategy helps drastically restructure the network and escape from local minima. We may also benefit from the well-developed AIG flows because AIG optimization has been researched for a longer time and by a broader range of developers such that AIG-based algorithms might have better performance and efficiency.

10.3 Design Space Exploration for AQFP Technology Mapping

Imagine a design space consisting of all legal and logically equivalent mapped networks, the optimization problem of AQFP technology mapping is to find the best one in the design space in terms of a cost metric (usually, JJ count or depth). Performing MIG restructuring and AQFP legalization can be seen as moving along two orthogonal directions (or axes) in the design space, exploring first different logically-equivalent MIGs without buffers, and then different mapped networks corresponding to the same MIG. This approach confines the degree of freedom of the exploration in order to be more scalable and potentially explore a larger space within the confined regions. However, if the two axes are only explored once each, then still only a small subset of the entire space is explored and the result may be far from the global optimal. The major problem is that during MIG restructuring, buffers are not inserted yet and the algorithm can only decide on the best moves based on a truncated cost metric (usually, MIG size or depth) which does not completely correlate to the actual cost metric.

Algorithm 10.1: AQFP technology mapping with design space exploration (solves Problem 1)

```
Input: Unconstrained network $N$
Output: Optimized mapped network $N'$
1  $N_0^* \leftarrow$ map_into_MIG($N$)                                   // [Tem+22]
2  $N_{best}^* \leftarrow$ copy($N_0^*$)
3  $best\_cost \leftarrow \infty$
4  for $restart = 1$ upto $num\_restarts$ do
5      $N_{best\_inner}^* \leftarrow N_0^*$
6      $N_{curr}^* \leftarrow N_0^*$
7      $best\_cost\_inner \leftarrow \infty$
8      $rnd \leftarrow$ new_random_engine()
9      $timer \leftarrow$ start_timer()
10     for $step = 1$ upto $max\_steps$ do
11         $N_{curr}^* \leftarrow$ restructure_MIG_randomly($N_{curr}^*$, $rnd$)   // [MCB07; Tem+22; LRD21; AGD16; Mis+11a]
12         $curr\_cost \leftarrow$ evaluate(legalize($N_{curr}^*$))              // alg. 9.6
13         if $curr\_cost < best\_cost\_inner$ then
14             $N_{best\_inner}^* \leftarrow N_{curr}^*$
15             $best\_cost\_inner \leftarrow curr\_cost$
16             $last\_impr \leftarrow step$
17         if $step - last\_impr \geq max\_no\_impr$ then break
18         if elapsed_time($timer$) $\geq timeout$ then break
19     if $best\_cost\_inner < best\_cost$ then
20         $N_{best}^* \leftarrow N_{best\_inner}^*$
21         $best\_cost \leftarrow best\_cost\_inner$
22 $N' \leftarrow$ legalize($N_{best}^*$)                                  // alg. 9.6
23 return $N'$
```

We propose to use the design space exploration approach described in Chapter 5 for AQFP technology mapping. The flow is illustrated in Algorithm 10.1, which performs multiple iterations of MIG restructuring and legalizes the MIGs in every iteration to compute the actual JJ cost, such that the exploration is correctly guided. As formulated in Problem 1, the input is an unconstrained network N, so we first map it into an MIG network (line 1). In the rest of the algorithm, four copies of the MIG are maintained: the initial MIG N_0^*, the overall best MIG N_{best}^*, the best MIG in the inner for-loop $N_{\text{best\_inner}}^*$, and the current MIG N_{curr}^*. The algorithm explores the design space by starting *num_restarts* times from the initial point N_0^* (the outer for-loop, lines 4-21), each time exploring MIGs along a random trajectory (the inner for-loop, lines 10-18). For each MIG, the second axis of different mapped networks is also explored, and the cost is evaluated on the best mapped network (line 12). The best-seen MIGs are book-marked on the current trajectory ($N_{\text{best\_inner}}^*$, line 14) and on all trajectories (N_{best}^*, line 20). The inner loop is terminated when no improvement is observed for *max_no_impr* steps consecutively (line 17), or when the timeout limit is exceeded (line 18).

The key ingredients are the functions map_into_MIG (line 1), restructure_MIG_randomly (line 11), and legalize (lines 12 and 22). In line 1, function map_into_MIG calls a graph mapping algorithm [Tem+22]. In the case where N is an AIG, it can also be transformed directly into an MIG by converting each AND2 into a MAJ3 with a constant 0 fanin. In line 11, function restructure_MIG_randomly applies a randomly-chosen MIG restructuring script. In our experience, scripts that perform well consist of a drastic restructuring step, such as mapping into k-LUT network [MCB07] and then remapping into MIG [Tem+22], followed by some MIG optimization steps, such as resubstitution [LRD21; LM23], algebraic rewriting [AGD16], and balancing [Mis+11a]. In line 12, the current MIG is legalized using the proposed legalization flow (Algorithm 9.6) to obtain a mapped network N' for evaluation. Depending on the design objective, the function evaluate may return the JJ count (#JJs = $6 \cdot |G'| + 2 \cdot |B|$), depth ($d(N')$), or energy-delay product (EDP = #JJs $\cdot\, d(N')$). Line 22 legalizes the best MIG again also using Algorithm 9.6. If better runtime efficiency is desired, lighter-effort legalization (for example, by limiting the number of optimization iterations in Algorithm 9.5) can be used in line 12 for cost evaluation while keeping the final legalization in line 22 the highest-affordable effort.

The advantages of this design space exploration approach are two-fold. First, compared to existing approaches, it explores a larger design space, and the frontier of exploration also stretches further. This is thanks to the hill-climbing strategy, where we simply record the best-seen design on the trajectory and keep moving forward when the cost gets worse instead of rolling back. Moreover, the key enabling factor to explore on the orthogonal axis (different mapped networks from the same MIG) is that the legalization runtime is fast enough, which motivates the focus of this paper on efficient heuristic buffer optimization methods instead of unscalable exact algorithms. The second advantage of Algorithm 10.1 is that the design space exploration is done on the fly. That is, no heavy data training, complicated decision-making, or human expert intuition is needed to guide the exploration, and the results are not over-fitted for a subset of benchmarks. The direction of exploration is guided by the simplest strategy, randomness, and the best transformation sequence is discovered on the fly. As there is a factor of luck involved, the purpose of the outer loop is to mitigate the possibility of a “bad” random seed leading to unsatisfactory results and to increase the chance of meeting at least one “good” random sequence in all restarts.

10.4 Verification

To ensure the correct functionality of the synthesized AQFP circuit, two types of verification should be performed: logic equivalence to the specification and legality with respect to the AQFP technology constraints. These correspond to the first and the third condition in Problem 1. The second condition, i.e., only AQFP-compatible gates are used, is ensured automatically by having used MIG as logic representation in the restructuring step.

For logic equivalence, we apply the well-developed combinational equivalence checking algorithm [Mis+06a] on the mapped network N' and the original network N. For legality verification, we check if the mapped network is indeed path-balanced and properly-branched. First, a schedule $\mathcal{S}$ of the mapped network is (re-)computed by visiting all nodes in a topological

Table 10.1: Best-known results on AQFP technology mapping.

Bench.	SoTA [Fu+23b]			Ours (Algorithm 10.1)							
	#JJs	$d(N')$	EDP	#JJs		$d(N')$		EDP		Time (s)	Eval. (s)
5xp1	726	10	7260	368	-49%	9	-10%	3312	-54%	66.2	0.7
c1908	5108	34	173672	4434	-13%	29	-15%	128586	-26%	190.4	36.4
c432	3098	34	105332	2342	-24%	27	-21%	63234	-40%	68.0	2.7
c5315	16410	30	492300	13986	-15%	24	-20%	335664	-32%	519.8	267.8
c880	3876	23	89148	3364	-13%	19	-17%	63916	-28%	100.6	14.6
chkn	3500	15	52500	2238	-36%	15	0%	33570	-36%	96.5	6.0
count	1400	12	16800	1302	-7%	11	-8%	14322	-15%	77.3	1.4
dist	3536	14	49504	1824	-48%	14	0%	25536	-48%	116.7	6.1
in5	3370	14	47180	1602	-52%	13	-7%	20826	-56%	120.2	4.4
in6	2884	11	31724	1708	-41%	12	+9%	20496	-35%	90.3	3.5
k2	14748	22	324456	8376	-43%	19	-14%	159144	-51%	404.7	102.8
m3	2680	12	32160	1600	-40%	12	0%	19200	-40%	115.6	4.3
max512	4812	16	76992	2740	-43%	14	-13%	38360	-50%	140.8	10.1
misex3	11272	20	225440	2634	-77%	17	-15%	44778	-80%	238.1	21.9
mlp4	2976	14	41664	1588	-47%	14	0%	22232	-47%	160.0	7.3
prom2	22326	20	446520	15258	-32%	16	-20%	244128	-45%	788.8	286.5
sqr6	916	10	9160	710	-22%	9	-10%	6390	-30%	59.3	0.7
x1dn	1208	11	13288	714	-41%	10	-9%	7140	-46%	61.5	0.5
Total	104846	322	2235100	66788	-36%	285	-12%	1239208	-44%	3414.6	777.8

order and assigning:

$$\mathcal{S}(n) = \begin{cases} 0 & \text{if } n \in I \\ \max\limits_{n_i \in \mathrm{FI}(n)} \mathcal{S}(n_i) + 1 & \text{otherwise.} \end{cases} \tag{10.1}$$

Then, we verify if N' is path-balanced by traversing all nodes again and testing Equations (9.1) to (9.3). The "for all edges" in Equation (9.1) is equivalent to checking all fanins n_1 of all gates n_2. Finally, we verify if N' is properly branched by comparing the number of fanouts of all PIs, gates, and buffers against the parameters s_i, s_g, and s_b, respectively. With our data structure and constraint formulation, the AQFP technology legality verification can be done in linear time.

10.5 Experimental Results

In this section, we present the experimental results of Algorithm 10.1 on solving Problem 1. To be consistent with previous works that we compare to, we use $s_b = 4$ for the splitting capacity of buffers.

With the proposed design space exploration approach presented in Section 10.3, we present new best-known results on the problem of AQFP technology mapping on the MCNC benchmark suite [Yan91]. In Table 10.1, our results are compared to SoTA [Fu+23b]. Since [Fu+23b] outperformed other previous works [Cai+19b; Tes+21; MRM21; Meu+22] on all benchmarks and on all metrics[1], data from these works is omitted. We use the same optimization objective as in [Fu+23b], i.e., minimizing *energy-delay product* (EDP). The parameters used in Algorithm 10.1 are *num_restarts* = 5, *max_steps*[2] = 1000, *max_no_impr* = 50, and *timeout* = 100 seconds.

In addition to #JJs, $d(N')$ and EDP, the last two columns in Table 10.1 list, respectively, the total runtime of Algorithm 10.1 (column "Time") and the runtime for cost evaluation (line 12 in Algorithm 10.1, column "Eval.") using Algorithm 9.6. The runtime information of [Fu+23b] is unfortunately not provided.

Our design space exploration achieves strictly better results than [Fu+23b] in #JJs and EDP on all benchmarks. In total, 36% improvement in #JJs, 12% improvement in depth, and 44% improvement in EDP are achieved within manageable runtime.

10.6 Summary

This chapter collects various elements presented in this thesis as a complete AQFP logic synthesis toolbox. First, as AQFP is based on majority gates, we combine the simulation-guided paradigm introduced in Chapter 3 with the MAJ-based resynthesis algorithm proposed in Chapter 4 as a high-effort MIG resubstitution algorithm. Together with other existing algorithms, a portfolio of MIG restructuring commands is established. Then, we discuss whether logic optimization and technology legalization should be tackled together or separately and argue that when circuits are small enough or the runtime budget is sufficient, the two should be interleaved to achieve better results. For such, we leverage the unsupervised design space exploration framework proposed in Chapter 5 and combine it with the AQFP legalization flow presented in Chapter 9. Experimental results show that our AQFP technology mapping methodology gives a significant 44% improvement in the energy-delay product compared to the best-known AQFP synthesis results. For the sake of completeness, we also discuss verification methods for legalized AQFP circuits. All the presented experimental results are verified and published[3] for third-party verification.

[1] [Tes+21] and [MRM21] used different assumptions, i.e. primary inputs do not need to be balanced, so the numbers presented in the papers are different. As both works are open-sourced and flexible to taking different assumptions, we reran the experiment with the same assumptions for a fair comparison.

[2] All restarts end within 200 steps due to the two terminating conditions, so this value is never really reached.

[3] https://github.com/lsils/SCE-benchmarks

11 Conclusions

Logic synthesis is a field of intractable problems with heuristic solutions. It is a story of the mutual stimulation between the scaling of computing systems and the advancement of synthesis algorithms. Motivated by the need for higher-performance logic synthesis and the unconventional challenges posed by emerging technologies, we presented in this thesis various breakthroughs in contemporary logic synthesis and an in-depth investigation into the problem of AQFP legalization, with a closing demonstration in Chapter 10 showing an application of contemporary logic synthesis techniques in AQFP circuit optimization.

In this section, we first summarize important technical and experimental results presented in each chapter. Then, an overview of the most significant contributions of this thesis is given, along with a discussion on future perspectives and open problems.

11.1 Summary of Important Results

- **Chapter 3: Simulation-Guided Paradigm**
  - With experiments on various simulation pattern generation strategies, we found that the strategy "`rand 256 + 1x s-a-obs`" (i.e., starting with 256 random patterns and generating at least one stuck-at pattern for each node with consideration of observability) performs the best in generating expressive patterns that reduce 99.5% counter-examples encountered in simulation-guided resubstitution.
  - By generating expressive simulation patterns, instead of using random patterns, runtime is shifted from optimization (resubstitution) to pattern generation. If these patterns are pre-generated and reused, then this means optimization time is highly reduced.
  - Using ECO benchmarks, we showed that the simulation patterns and the counter-examples are still effective on functionally modified benchmarks in reducing runtime.
  - The simulation-guided resubstitution is capable of using a much larger window

size and achieves a 5.9% reduction in the number of AIG nodes, compared to 3.7% by a state-of-the-art resubstitution algorithm, within comparable runtime.

- **Chapter 4: Heuristic Resynthesis**
  - High-effort resubstitution reduces AIG, XAG, MIG, and MuxIG sizes by an additional 1.77%, 2.86%, 2.45%, and 20.24%, respectively, on highly optimized (saturated) benchmarks within smaller or similar runtime.
  - The proposed high-effort heuristic resynthesis algorithms have better complexities compared to existing approaches. The AND-based resynthesis algorithm has $\mathcal{O}(n^2 ml)$ complexity and the MAJ- and MUX-based resynthesis algorithms have $\mathcal{O}(nml)$ complexity, where n is the number of divisors, m is the number of gates in dependency circuit, and l is the length of truth tables (simulation signatures).
- **Chapter 5: Design Space Exploration**
  - We presented new best results on the problem of MIG size optimization, which are better than or the same as the state-of-the-art on all benchmarks with an improvement of 16.9% on average.
- **Chapter 6: Testing and Debugging Logic Synthesis Algorithms**
  - We adapted the fuzz testing technique to generate logic networks for the testing of logic synthesis algorithms. Our topology-based fuzzer captures defects in *ABC*, *mockturtle* and *LSOracle* using 93% smaller testcases compared to an existing AIG fuzzer `aigfuzz`.
  - A testcase minimizer specialized for logic networks was developed based on the delta debugging technique. Our minimizer isolates smaller or equal-sized minimal failure-inducing cores using 50% oracle calls and 50% runtime compared to an existing AIG delta debugger `aigdd`.
- **Chapter 8: Impact of Sequential Design on AQFP Technology Constraints**
  - We re-examined the formulation of AQFP technology constraints and propose possible relaxations on these constraints: phase alignment instead of path balancing, and the flexibilities on combinational inputs' splitting capacity and phases. However, phase alignment comes with a tradeoff on the possibility of wave-pipelining.
  - Adopting the relaxed constraints reduces 73% of buffers on average, and up to 90% in some particularly-imbalanced benchmarks.
- **Chapter 9: AQFP Technology Legalization by Buffer/Splitter Insertion**
  - We presented a heuristic AQFP legalization and optimization flow that obtains similar, near-optimal quality as the state-of-the-art ILP-based algorithm within very little runtime.

  - Our scheduling-based AQFP legalization approach is fast and scalable. We demonstrate legalization results on benchmarks 10× to 100× larger than what any other related works could handle.

- **Chapter 10: AQFP Logic Synthesis Toolbox**
  - We presented AQFP technology mapping results strictly better than the state-of-the-art in #JJs and EDP on all benchmarks. In total, 36% improvement in #JJs, 12% improvement in depth, and 44% improvement in EDP are achieved within manageable runtime.

11.2 Thesis Contributions

The three most important contributions of this thesis are:

- **Chapters 3 and 4 — High-effort simulation-guided resubstitution [Lee+22; LM23].** Combining the simulation-guided paradigm and high-effort resynthesis in a resubstitution algorithm, we provide an opportunity to keep optimizing benchmarks that are already highly optimized. The simulation-guided paradigm allows us to enlarge the window size and unlocks global consideration of don't cares. The heuristic resynthesis algorithms are unlimited in the size of dependency circuits, broadening the search space and finding more optimization opportunities.
- **Chapters 8 and 9 — Pioneering investigation on the problem of AQFP legalization [LAD23; Lee+24].** By really diving into the details of AQFP systems design and circuit properties, we set the ground for realistic directions of research in the AQFP legalization problem. By establishing a linear relation between a schedule and the minimum buffer count, we identify that the AQFP legalization problem is a scheduling problem. Built upon this observation, we presented an AQFP legalization flow consisting of depth-optimal scheduling, irredundant buffer insertion, and heuristic optimization.
- **Chapters 5 and 10 — Significant improvements in AQFP circuit optimization [LRD23; Lee+24].** We presented an AQFP technology mapping flow combining on-the-fly design space exploration, simulation-guided MIG resubstitution, and AQFP legalization, revisiting various elements in this thesis. A significant 44% improvement in EDP was achieved, demonstrating the power of advanced logic synthesis techniques presented in this thesis.

11.3 Open Problems

In addition to future works directly related to the research problem or ideas in each chapter that we have discussed at the end of the respective chapter, we list here the open problems in a broader sense as possible future research directions.

11.3.1 Endless Pursuit for QoR and Efficiency

Logic synthesis problems are mostly NP-hard. Unless P=NP is proven or quantum computing becomes realistic, all we can do is develop better and better heuristics. This is an endless competition with never-satisfying results. Thus, in some sense, we could say that logic synthesis never dies as an ongoing research field. It may become more and more difficult, but new algorithms or strategies getting another percent QoR improvement are likely always possible and companies are willing to pay the extra synthesis runtime for it. Conversely, if the efficiency of logic synthesis algorithms improves and the same QoR could be achieved within less runtime, then more optimization iterations or higher-effort parameters could be applied and better QoR could be achieved within the same runtime budget.

11.3.2 AQFP Synthesis: Integration into Production-Ready EDA Tools

Research and development of EDA for AQFP are still in a relatively early stage. Various algorithms are independently developed by different research groups in different EDA systems. The Cadence system with a complete, working flow used for fabrication by the lab at Yokohama National University (where AQFP was first proposed) does not adopt all the latest algorithms yet. Due to the special clock phase assignment issue, common file-exchange formats need to be extended, and such an extended format has to be agreed upon by developers of different algorithms in order for them to be compatible. Moreover, as discussed in Chapter 8, not all of the newly proposed algorithms respect the actual properties and realistic constraints of the technology, and some subtle constraints can only be correctly considered with communication and collaboration between algorithms in different synthesis stages. Hence, it is important to make an effort to integrate all state-of-the-art algorithms into one EDA tool.

11.3.3 Other Emerging Technologies

In this thesis, we take the optimization of AQFP circuits as an example application of contemporary logic synthesis techniques. There are many more emerging technologies and computing paradigms that are being rapidly developed and shown to be promising. Each of them has different properties and constraints to be considered in logic synthesis. For example, technologies in the *field-coupled nanocomputing* (FCN) family require the circuit to be planarized by inserting crossing cells, in addition to similar path-balancing and fanout-branching constraints as AQFP [Wal+19]. As the crossing cells also need to be balanced, the three constraints must be considered together, making it a similar but harder problem than the AQFP legalization problem.

11.4 Final Remarks

All of the algorithms, frameworks, and flows presented in this thesis are implemented in the open-source C++ logic synthesis library *mockturtle*[1] [Rie+19b; Soe+22]. Whenever possible, verification is performed so that correct results are ensured.

[1]https://github.com/lsils/mockturtle

Bibliography

[AA20] Bijan Alizadeh and Yasaman Abadi. “Incremental SAT-based correction of gate level circuits by reusing partially corrected circuits”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12 (2020), pp. 3063–3067.

[AB14] Neal G. Anderson and Sanjukta Bhanja, eds. *Field-Coupled Nanocomputing - Paradigms, Progress, and Perspectives*. Vol. 8280. Lecture Notes in Computer Science. Springer, 2014.

[ACY19] Christopher L Ayala, Olivia Chen, and Nobuyuki Yoshikawa. “AQFPTX: Adiabatic Quantum-Flux-Parametron Timing eXtraction Tool”. In: *2019 IEEE International Superconductive Electronics Conference (ISEC)*. 2019, pp. 1–3.

[AGD15] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “The EPFL combinational benchmark suite”. In: *Proceedings of IWLS*. 2015.

[AGD16] Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. “Majority-Inverter Graph: A New Paradigm for Logic Optimization”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35.5 (2016), pp. 806–819.

[Ake62] Sheldon B. Akers Jr. “Synthesis of combinational logic using three-input majority gates”. In: *3rd Annual Symposium on Switching Circuit Theory and Logical Design*. 1962, pp. 149–157.

[Ake78] Sheldon B. Akers Jr. “Binary Decision Diagrams”. In: *IEEE Trans. Computers* 27.6 (1978), pp. 509–516.

[Ama+18] Luca Gaetano Amarù et al. “Improvements to Boolean resynthesis”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018*. 2018, pp. 755–760.

[Aya+15] Christopher L Ayala et al. “Timing Extraction for Logic Simulation of VLSI Adiabatic Quantum-Flux-Parametron Circuits”. In: *IEICE Technical Report*. SCE2015-21 115.242 (Oct. 2015), pp. 7–12.

[Aya+17] Christopher L Ayala et al. “Majority-Logic-Optimized Parallel Prefix Carry Look-Ahead Adder Families Using Adiabatic Quantum-Flux-Parametron Logic”. In: *IEEE Trans. Appl. Supercond.* 27.4 (June 2017), pp. 1–7.

[Aya+20] Christopher L Ayala et al. "A semi-custom design methodology and environment for implementing superconductor adiabatic quantum-flux-parametron microprocessors". In: *Superconductor Science and Technology* 33.5 (2020).

[Aya+21] Christopher L. Ayala et al. "MANA: A Monolithic Adiabatic iNtegration Architecture Microprocessor Using 1.4-zJ/op Unshunted Superconductor Josephson Junction Devices". In: *IEEE J. Solid State Circuits* 56.4 (2021), pp. 1152–1165.

[Bar+88] Karen A. Bartlett et al. "Multi-level logic minimization using implicit don't cares". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 7.6 (1988), pp. 723–740.

[BD97] Valeria Bertacco and Maurizio Damiani. "The disjunctive decomposition of logic functions". In: *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1997.* 1997, pp. 78–82.

[BHS90] Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. "Multilevel logic synthesis". In: *Proceedings of IEEE* 78.2 (1990), pp. 264–300.

[Bie+09] Armin Biere et al., eds. *Handbook of Satisfiability.* Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[BM06] Robert Brayton and Alan Mishchenko. "Scalable logic synthesis using a simple circuit structure". In: *IWLS 2006.* Vol. 6. 2006, pp. 15–22.

[BM10] Robert K. Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Proceedings of CAV.* 2010, pp. 24–40.

[Boo47] George Boole. *The mathematical analysis of logic.* Philosophical Library, 1847.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Proceedings of TACAS.* Vol. 9035. Springer, 2015, pp. 194–199.

[Bra+82] Robert K Brayton et al. "A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization". In: *Proceedings of the International Symposium on Circuits and Systems.* 1982, pp. 42–48.

[Bra+87] Robert K. Brayton et al. "MIS: A Multiple-Level Logic Optimization System". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 6.6 (1987), pp. 1062–1081.

[Bra82] Robert K Brayton. "The decomposition and factorization of Boolean expressions". In: *ISCA-82* (1982), pp. 49–54.

[Bra83] Daniel Brand. "Redundancy and Don't Cares in Logic Synthesis". In: *IEEE Trans. Computers* 32.10 (1983), pp. 947–952.

[Bur+98] Wayne P. Burleson et al. "Wave-pipelining: a tutorial and research survey". In: *IEEE Trans. Very Large Scale Integr. Syst.* 6.3 (1998), pp. 464–474.

[Cai+19a] Ruizhe Cai et al. "A Buffer and Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits". In: *Proceedings of ICCD.* 2019, pp. 429–436.

[Cai+19b] Ruizhe Cai et al. “A Majority Logic Synthesis Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits”. In: *Proceedings of GLSVLSI.* 2019, pp. 189–194.

[Cai+19c] Ruizhe Cai et al. “IDE Development, Logic Synthesis and Buffer/Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits”. In: *Proceedings of ISVLSI.* IEEE. 2019, pp. 187–192.

[CD23] Alessandro Tempia Calvino and Giovanni De Micheli. “Depth-Optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits”. In: *Proceedings of ASPDAC.* 2023, pp. 152–158.

[CD94a] Jason Cong and Yuzheng Ding. “FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 13.1 (1994), pp. 1–12.

[CD94b] Jason Cong and Yuzheng Ding. “On area/depth trade-off in LUT-based FPGA technology mapping”. In: *IEEE Trans. Very Large Scale Integr. Syst.* 2.2 (1994), pp. 137–148.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *OSDI 2008.* USENIX Association, 2008, pp. 209–224.

[Cha+06] Satrajit Chatterjee et al. “Reducing structural bias in technology mapping”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 25.12 (2006), pp. 2894–2903.

[Chu+18] Zhufei Chu et al. “Functional decomposition using majority”. In: *Proceedings of ASP-DAC.* IEEE. 2018, pp. 676–681.

[Cla+00] Edmund M. Clarke et al. “Counterexample-Guided Abstraction Refinement”. In: *Proceedings of CAV.* Vol. 1855. 2000, pp. 154–169.

[CM10] Jason Cong and Kirill Minkovich. “LUT-based FPGA technology mapping for reliability”. In: *Proceedings of DAC.* 2010, pp. 517–522.

[CPD96] Jason Cong, John Peck, and Yuzheng Ding. “RASP: A general logic synthesis system for SRAM-based FPGAs”. In: *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays.* 1996, pp. 137–143.

[CR88] Henry Cox and Janusz Rajski. “Stuck-open and transition fault testing in CMOS complex gates”. In: *Proceedings of of ITC.* IEEE. 1988, pp. 688–694.

[Cra57] William Craig. “Linear reasoning: A new form of the Herbrand-Gentzen theorem”. In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 250–268.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture - a hardware / software approach.* 1999. ISBN: 978-1-55860-343-1.

[CWB15] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-Adaptive Mutational Fuzzing”. In: *SP 2015.* IEEE Computer Society, 2015, pp. 725–741.

[DD90] Maurizio Damiani and Giovanni De Micheli. “Observability Don’t Care Sets and Boolean Relations”. In: *Proceedings of ICCAD*. 1990, pp. 502–505.

[De 94] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[DM93] Maurizio Damiani and Giovanni De Micheli. “Don’t care set specifications in combinational and synchronous logic circuits”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 12.3 (1993), pp. 365–388.

[Don+22] Peiyan Dong et al. “TAAS: a timing-aware analytical strategy for AQFP-capable placement automation”. In: *Proceedings of DAC*. 2022, pp. 1321–1326.

[DRH14] Kyle Dewey, Jared Roesch, and Ben Hardekopf. “Language fuzzing using constraint logic programming”. In: *ASE 2014*. ACM, 2014, pp. 725–730.

[FDK11] Petra Färm, Elena Dubrova, and Andreas Kuehlmann. “Integrated logic synthesis using simulated annealing”. In: *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*. 2011, pp. 407–410.

[Fis+10] Petr Fiser et al. “On logic synthesis of conventionally hard to synthesize circuits using genetic programming”. In: *Proceedings of DDECS*. 2010, pp. 346–351.

[Fu+23a] Rongliang Fu et al. “A Global Optimization Algorithm for Buffer and Splitter Insertion in Adiabatic Quantum-Flux-Parametron Circuits”. In: *Proceedings of ASPDAC*. 2023, pp. 769–774.

[Fu+23b] Rongliang Fu et al. “BOMIG: A Majority Logic Synthesis Framework for AQFP Logic”. In: *Proceedings of DATE*. 2023.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing”. In: *NDSS 2008*. The Internet Society, 2008.

[Got59] Eiichi Goto. “The parametron, a digital computing element which utilizes parametric oscillation”. In: *Proceedings of the IRE* 47.8 (1959), pp. 1304–1316.

[Gro+22] Antoine Grosnit et al. “BOiLS: Bayesian Optimisation for Logic Synthesis”. In: *Proceedings of DATE*. IEEE, 2022, pp. 1193–1196.

[Haa+17] Winston Haaswijk et al. “A novel basis for logic rewriting”. In: *Proceedings of ASP-DAC*. 2017, pp. 151–156.

[Haa+18] Winston Haaswijk et al. “Integrated ESOP Refactoring for Industrial Designs”. In: *25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018*. 2018, pp. 369–372.

[Haa+20] Winston Haaswijk et al. “SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.4 (2020), pp. 871–884.

[Hac+89] G Hachtel et al. “BOLD: The Boulder optimal logic design system”. In: *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*. Vol. 1. IEEE Computer Society. 1989, pp. 59–60.

[Har+87] Yutaka Harada et al. “Basic operations of the quantum flux parametron”. In: *IEEE Trans. Magn.* 23.5 (1987), pp. 3801–3807.

[He+22] Yuxing He et al. “Low clock skew superconductor adiabatic quantum-flux-parametron logic circuits based on grid-distributed blocks”. en. In: *Supercond. Sci. Technol.* 36.1 (Dec. 2022), p. 015006.

[HFS17] Ivo Háleček, Petr Fišer, and Jan Schmidt. “Are XORs in logic synthesis really necessary?” In: *Proceedings of of DDECS.* 2017, pp. 134–139.

[HLH91] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu. “A formal approach to the scheduling problem in high level synthesis”. In: *IEEE Trans. on CAD* 10.4 (1991), pp. 464–475.

[HMB07] Aaron P. Hurst, Alan Mishchenko, and Robert K. Brayton. “Fast Minimum-Register Retiming via Binary Maximum-Flow”. In: *Proceedings of FMCAD.* 2007.

[Hos+91] Mutsumi Hosoya et al. “Quantum flux parametron: a single quantum flux device for Josephson supercomputer”. In: *IEEE Transactions on Applied Superconductivity* 1.2 (1991), pp. 77–89.

[Hua+21] Chao-Yuan Huang et al. “An Optimal Algorithm for Splitter and Buffer Insertion in Adiabatic Quantum-Flux-Parametron Circuits”. In: *Proceedings of ICCAD.* 2021.

[Jar+11] TAW Jarratt et al. “Engineering change: an overview and perspective on the literature”. In: *Research in Engineering Design* 22.2 (2011), pp. 103–124.

[KA22] Angshuman Khan and Rajeev Arya. “Design and energy dissipation analysis of simple QCA multiplexer for nanocomputing”. In: *J. Supercomput.* 78.6 (2022), pp. 8430–8444.

[KJR20] Victor N Kravets, Jie-Hong R Jiang, and Heinz Riener. “Learning to automate the design updates from observed engineering changes in the chip development cycle”. In: *Proceedings of DATE.* IEEE. 2020, pp. 738–743.

[KK04] Victor N Kravets and Prabhakar Kudva. “Implicit enumeration of structural changes in circuit optimization”. In: *Proceedings of DAC.* 2004, pp. 438–441.

[KL70] Robert W Keyes and Rolf Landauer. “Minimal energy dissipation in logic”. In: *IBM Journal of Research and Development* 14.2 (1970), pp. 152–157.

[KNP21] Gereon Kremer, Aina Niemetz, and Mathias Preiner. “ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends”. In: *CAV 2021.* Springer, 2021, pp. 231–242.

[Knu11] Donald Ervin Knuth. *The art of computer programming, volume 4A: combinatorial algorithms, part 1.* Addison-Wesley, 2011.

[KP18] Naveen Kumar Katam and Massoud Pedram. “Logic optimization, complex cell design, and retiming of single flux quantum circuits”. In: *IEEE Trans. Appl. Supercond.* 28.7 (2018), pp. 1–9.

[KS98] Victor N Kravets and Karem A Sakallah. “M32: A constructive multilevel logic synthesis system”. In: *Proceedings of DAC.* 1998, pp. 336–341.

[Kue+02] Andreas Kuehlmann et al. “Robust Boolean reasoning for equivalence checking and functional property verification”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 21.12 (2002), pp. 1377–1394.

[LAD23] Siang-Yun Lee, Christopher L Ayala, and Giovanni De Micheli. “Impact of Sequential Design on The Cost of Adiabatic Quantum-Flux Parametron Circuits”. In: *IEEE Transactions on Applied Superconductivity* (2023). DOI: 10.1109/TASC.2023.3308408.

[Lee+07] Chih-Chun Lee et al. “Scalable exploration of functional dependency by interpolation and incremental SAT solving”. In: *2007 International Conference on Computer-Aided Design, ICCAD 2007.* 2007, pp. 227–233.

[Lee+17] Seungsoo Lee et al. “DELTA: A Security Assessment Framework for Software-Defined Networks”. In: *NDSS 2017.* The Internet Society, 2017.

[Lee+19] Siang-Yun Lee et al. “Enumeration of Minimum Fanout-Free Circuit Structures”. In: *Proceedings of IWLS.* 2019.

[Lee+22] Siang-Yun Lee et al. “A Simulation-Guided Paradigm for Logic Synthesis and Verification”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.8 (2022), pp. 2573–2586. DOI: 10.1109/TCAD.2021.3108704.

[Lee+24] Siang-Yun Lee et al. “Technology Legalization and Optimization for Adiabatic Quantum-Flux Parametron”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* (2024). (Under review).

[Li+22] Xi Li et al. “Multi-Phase Clocking for Multi-Threaded Gate-Level-Pipelined Superconductive Logic”. In: *Proceedings of ISVLSI.* 2022, pp. 62–67.

[LM23] Siang-Yun Lee and Giovanni De Micheli. “Heuristic Logic Resynthesis Algorithms at the Core of Peephole Optimization”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 42.11 (2023), pp. 3958–3971. DOI: 10.1109/TCAD.2023.3256341.

[LPP96] Yung-Te Lai, K-RR Pan, and Massoud Pedram. “OBDD-based function decomposition: Algorithms and implementation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.8 (1996), pp. 977–990.

[LRD21] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Logic Resynthesis of Majority-Based Circuits by Top-Down Decomposition”. In: *Proceedings of DDECS.* 2021, pp. 105–110.

[LRD22a] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “An Automated Testing and Debugging Toolkit for Gate-Level Logic Synthesis Applications”. In: *Proceedings of IWLS.* 2022.

[LRD22b] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Beyond local optimality of buffer and splitter insertion for AQFP circuits”. In: *Proceedings of DAC.* 2022, pp. 445–450.

[LRD23] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. “Customizable On-the-fly Design Space Exploration for Logic Optimization of Emerging Technologies”. In: *Proceedings of IWLS*. 2023.

[LS91a] Charles E. Leiserson and James B. Saxe. “Retiming Synchronous Circuitry”. In: *Algorithmica* 6.1–6 (1991), pp. 5–35.

[LS91b] Konstantin K Likharev and Vasilii K Semenov. “RSFQ logic/memory family: A new Josephson-junction technology for sub-terahertz-clock-frequency digital systems”. In: *IEEE Trans. Appl. Supercond.* 1.1 (1991), pp. 3–28.

[Man+21] Valentin J. M. Manès et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2312–2331.

[MB05] Alan Mishchenko and Robert K. Brayton. “SAT-Based Complete Don’t-Care Computation for Network Optimization”. In: *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany.* IEEE Computer Society, 2005, pp. 412–417.

[MB06] Alan Mishchenko and Robert Brayton. “Scalable logic synthesis using a simple circuit structure”. In: *Proceedings of IWLS*. 2006, pp. 15–22.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of TACAS*. Vol. 4963. Springer, 2008, pp. 337–340.

[MB90] Patrick C. McGeer and Robert K. Brayton. “The observability don’t-care set and its approximations”. In: *Proceedings of ICCD*. IEEE Computer Society, 1990, pp. 45–48.

[MCB06] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. “DAG-aware AIG rewriting: A fresh look at combinational logic synthesis”. In: *Proceedings of the 43rd Design Automation Conference, DAC 2006*. Ed. by Ellen Sentovich. 2006, pp. 532–535.

[MCB07] Alan Mishchenko, Satrajit Chatterjee, and Robert K Brayton. “Improvements to technology mapping for LUT-based FPGAs”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 26.2 (2007), pp. 240–253.

[McC56] Edward J McCluskey. “Minimization of Boolean functions”. In: *The Bell System Technical Journal* 35.6 (1956), pp. 1417–1444.

[McN61] Robert McNaughton. “Unate Truth Functions”. In: *IRE Trans. Electron. Comput.* 10.1 (1961), pp. 1–6.

[MD23] Dewmini Sudara Marakkalage and Giovanni De Micheli. “Fanout-Bounded Logic Synthesis for Emerging Technologies”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 2023.

[Meu+22] Giulia Meuli et al. “Majority-based Design Flow for AQFP Superconducting Family”. In: *Proceedings of DATE*. IEEE, 2022, pp. 34–39.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44.

[Mis+05] Alan Mishchenko et al. “FRAIGs: A unifying representation for logic synthesis and verification”. In: *ERL Technical Report*. 2005.

[Mis+06a] Alan Mishchenko et al. “Improvements to combinational equivalence checking”. In: *Proceedings of ICCAD*. 2006, pp. 836–843.

[Mis+06b] Alan Mishchenko et al. “Using simulation and satisfiability to compute flexibilities in Boolean networks”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 25.5 (2006), pp. 743–755.

[Mis+11a] Alan Mishchenko et al. “Delay optimization using SOP balancing”. In: *Proceedings of ICCAD*. 2011, pp. 375–382.

[Mis+11b] Alan Mishchenko et al. “Scalable don’t-care-based logic optimization and resynthesis”. In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (2011), 34:1–34:23.

[MJV00] Julian F Miller, Dominic Job, and Vesselin K Vassilev. “Principles in the evolutionary design of digital circuits—Part I”. In: *Genetic programming and evolvable machines* 1.1 (2000), pp. 7–35.

[MK06] S. Mitra and K. S. Kim. “XPAND: an efficient test stimulus compression technique”. In: *IEEE Trans. Computers* 55.2 (2006), pp. 163–173.

[MRM21] Dewmini Sudara Marakkalage, Heinz Riener, and Giovanni De Micheli. “Optimizing Adiabatic Quantum-Flux-Parametron (AQFP) Circuits using an Exact Database”. In: *IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2021*. 2021, pp. 1–6.

[MS00] João P Marques-Silva and Karem A Sakallah. “Boolean satisfiability in electronic design automation”. In: *Proceedings of DAC*. 2000, pp. 675–680.

[MS06] Ghassan Misherghi and Zhendong Su. “HDD: Hierarchical Delta Debugging”. In: *ICSE 2006*. ACM, 2006, pp. 142–151.

[MSP01] Alan Mishchenko, Bernd Steinbach, and Marek Perkowski. “An algorithm for bi-decomposition of logic functions”. In: *Proceedings of DAC*. 2001, pp. 103–108.

[MTT61] Saburo Muroga, Iwao Toda, and Satoru Takasu. “Theory of majority decision elements”. In: *Journal of the Franklin Institute* 271.5 (1961), pp. 376–418.

[Mur+89] Saburo Muroga et al. “The Transduction Method-Design of Logic Networks Based on Permissible Functions”. In: *IEEE Trans. Computers* 38.10 (1989), pp. 1404–1424.

[NBG11] Dmitri E Nikonov, George I Bourianoff, and Tahir Ghani. “Proposal of a spin torque majority gate logic”. In: *IEEE Electron Device Letters* 32.8 (2011), pp. 1128–1130.

[Net+19] Walter Lau Neto et al. “LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper”. In: *ICCAD 2019*. ACM, 2019, pp. 1–6.

[Net+22] Walter Lau Neto et al. “FlowTune: End-to-end Automatic Logic Optimization Exploration via Domain-specific Multi-armed Bandit”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).

[New23] Apple Newsroom. *Apple introduces M2 Ultra.* [online] https://www.apple.com/newsroom/2023/06/apple-introduces-m2-ultra/. [Accessed: 05-01-2024]. 2023.

[NIO96] Chetana Nagendra, Mary Jane Irwin, and Robert Michael Owens. “Area-time-power tradeoffs in parallel adders”. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 43.10 (1996), pp. 689–702.

[OKR14] Hadi Owlia, Parviz Keshavarzi, and Abdalhossein Rezai. “A novel digital logic implementation approach on nanocrossbar arrays using memristor-based multiplexers”. In: *Microelectron. J.* 45.6 (2014), pp. 597–603.

[Per+21] Yasasvi V. Peruvemba et al. “RL-Guided Runtime-Constrained Heuristic Exploration for Logic Synthesis”. In: *ICCAD 2021*. IEEE, 2021, pp. 1–9.

[PP18] Ghasem Pasandi and Massoud Pedram. “PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits”. In: *IEEE Trans. Appl. Supercond.* 29.4 (2018), pp. 1–14.

[Qui52] Willard V Quine. “The problem of simplifying truth functions”. In: *The American mathematical monthly* 59.8 (1952), pp. 521–531.

[Ray+12] Sayak Ray et al. “Mapping into LUT structures”. In: *Proceedings of DATE*. 2012, pp. 1579–1584.

[Reg+12] John Regehr et al. “Test-case reduction for C compiler bugs”. In: *PLDI 2012*. ACM, 2012, pp. 335–346.

[Rie+18] Heinz Riener et al. “Size Optimization of MIGs with an Application to QCA and STMG Technologies”. In: *Proceedings of the 14th IEEE/ACM International Symposium on Nanoscale Architectures, NANOARCH 2018*. 2018, pp. 157–162.

[Rie+19a] Heinz Riener et al. “On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis”. In: *Proceedings of DATE*. 2019, pp. 1649–1654.

[Rie+19b] Heinz Riener et al. “Scalable Generic Logic Synthesis: One Approach to Rule Them All”. In: *DAC 2019*. ACM, 2019, p. 70.

[Rie+22] Heinz Riener et al. “Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis”. In: *Proceedings of ASPDAC*. 2022, pp. 395–402.

[RMS20] Heinz Riener, Alan Mishchenko, and Mathias Soeken. “Exact DAG-Aware Rewriting”. In: *Proceedings of DATE*. 2020, pp. 732–737.

[Rot66] J Paul Roth. “Diagnosis of automata failures: A calculus and a method”. In: *IBM Journal of Research and Development* 10.4 (1966), pp. 278–291.

[Sai+21] Ro Saito et al. “Logic synthesis of sequential logic circuits for adiabatic quantum-flux-parametron logic”. In: *IEEE Trans. Appl. Supercond* 31.5 (2021), pp. 1–5.

[Sat+91] Hitomi Sato et al. “Boolean resubstitution with permissible functions and binary decision diagrams”. In: *Proceedings of DAC*. 1991, pp. 284–289.

[SAY21] Ro Saito, Christopher L Ayala, and Nobuyuki Yoshikawa. “Buffer reduction via N-phase clocking in adiabatic quantum-flux-parametron benchmark circuits”. In: *IEEE Trans. Appl. Supercond.* 31.6 (2021), pp. 1–8.

[SB00] Christoph Scholl and Bernd Becker. “On the Generation of Multiplexer Circuits for Pass Transistor Logic”. In: *2000 Design, Automation and Test in Europe DATE 2000.* 2000, pp. 372–378.

[Sch78] Thomas J Schaefer. “The complexity of satisfiability problems”. In: *Proceedings of ACM Symposium on Theory of Computing.* 1978, pp. 216–226.

[Sen+92] E.M. Sentovich et al. *SIS: A System for Sequential Circuit Synthesis.* Tech. rep. EECS Department, University of California, Berkeley, 1992.

[Sha38] Claude E Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pp. 713–723.

[SK04] Nikhil Saluja and Sunil P. Khatri. “A robust algorithm for approximate compatible observability don’t care (CODC) computation”. In: *Proceedings of DAC.* 2004, pp. 422–427.

[Soe+22] Mathias Soeken et al. *The EPFL Logic Synthesis Libraries.* 2022. arXiv: 1805.05121. URL: http://arxiv.org/abs/1805.05121.

[Tak+13] Naoki Takeuchi et al. “An adiabatic quantum flux parametron as an ultra-low-power logic device”. In: *Superconductor Science and Technology* 26.3 (2013), p. 035010.

[Tak+17] Naoki Takeuchi et al. “Adiabatic quantum-flux-parametron cell library designed using a 10 kA cm^{-2} niobium fabrication process”. In: *Superconductor Science and Technology* 30.3 (2017), p. 035002.

[Tak+19] Naoki Takeuchi et al. “Low-latency adiabatic superconductor logic using delay-line clocking”. In: *Applied Physics Letters* 115.7 (2019), p. 072601.

[TD24] Alessandro Tempia Calvino and Giovanni De Micheli. “Scalable Logic Rewriting Using Don’t Cares”. In: *Proceedings of DATE.* 2024.

[Tem+22] Alessandro Tempia Calvino et al. “A Versatile Mapping Approach for Technology Mapping and Graph Optimization”. In: *27th Asia and South Pacific Design Automation Conference, ASP-DAC 2022.* 2022, pp. 410–416.

[Tes+21] Eleonora Testa et al. “Algebraic and Boolean Optimization Methods for AQFP Superconducting Circuits”. In: *Proceedings of ASP-DAC.* 2021, pp. 779–785.

[Tov84] Craig A Tovey. “A simplified NP-complete satisfiability problem”. In: *Discrete Applied Mathematics* 8.1 (1984), pp. 85–89.

[Tse83] Grigori S Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of reasoning.* Springer, 1983, pp. 466–483.

[Tsu+17] Naoki Tsuji et al. “Design and implementation of a 16-word by 1-bit register file using adiabatic quantum flux parametron logic”. In: *IEEE Transactions on Applied Superconductivity* 27.4 (2017), pp. 1–4.

[TYY15] Naoki Takeuchi, Yuki Yamanashi, and Nobuyuki Yoshikawa. “Adiabatic quantum-flux-parametron cell library adopting minimalist design”. In: *Journal of Applied Physics* 117.17 (2015), p. 173912.

[Wal+19] Marcel Walter et al. “Scalable design for field-coupled nanocomputing circuits”. In: *Proceedings of ASP-DAC*. 2019.

[Xu+17] Qiuyun Xu et al. “Synthesis flow for cell-based adiabatic quantum-flux-parametron structural circuit generation with HDL back-end verification”. In: *IEEE Transactions on Applied Superconductivity* 27.4 (2017), pp. 1–5.

[Yan91] Saeyang Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

[YC02] Congguang Yang and Maciej J. Ciesielski. “BDS: a BDD-based logic optimization system”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 21.7 (2002), pp. 866–876.

[YCM17] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. “Fast algebraic rewriting based on and-inverter graphs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (2017), pp. 1907–1911.

[YTY21] Taiki Yamae, Naoki Takeuchi, and Nobuyuki Yoshikawa. “Adiabatic quantum-flux-parametron with delay-line clocking: logic gate demonstration and phase skipping operation”. In: *Superconductor Science and Technology* 34.12 (2021), p. 125002.

[YXM18] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. “Developing Synthesis Flows without Human Knowledge”. In: *Design Automation Conference (DAC’18)* (2018).

[Zel99] Andreas Zeller. “Yesterday, My Program Worked. Today, It Does Not. Why?” In: *SIGSOFT 1999*. Springer, 1999, pp. 253–267.

[ZH02] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Software Eng.* 28.2 (2002), pp. 183–200.

[Zog+17] Odysseas Zografos et al. “Wave pipelining for majority-based beyond-CMOS technologies”. In: *Proceedings of DATE*. 2017, pp. 1306–1311.

Lausanne, Switzerland
siang-yun.lee@epfl.ch
siangyun.sonia.lee@gmail.com

Siang-Yun Lee

Ph.D. Candidate

lee30sonia.github.io
github.com/lee30sonia
0000-0001-5907-2314

- Research interests: Logic synthesis, design automation for emerging technologies, computational neuroscience
- Current maintainer of the EPFL logic synthesis library mockturtle
- Experiences in reviewing, teaching, course design, and supervising student projects

Education & Work Experience

Software Engineer Internship, *Cadence Design Systems, Munich, Germany*	04/2023 — 09/2023
Doctoral Program in Computer and Communication Sciences, *École Polytechnique Fédéral de Lausanne (EPFL)*	2019 — 2024
Bachelor of Science in Electrical Engineering, *National Taiwan University (NTU)*	2015 — 2019

Research Projects & Selected Publications

Synthesis and Optimization for Emerging Technologies in Superconducting Electronics **08/2020 — 03/2024**
Advisor: Prof. Giovanni De Micheli *Integrated Systems Lab, EPFL*

- **Siang-Yun Lee**, Christopher L. Ayala, and Giovanni De Micheli. "Impact of Sequential Design on The Cost of Adiabatic Quantum-Flux Parametron Circuits," IEEE Trans. on Applied Superconductivity (TAS). 2023.
- **Siang-Yun Lee**, Heinz Riener, and Giovanni De Micheli. "Beyond Local Optimality of Buffer and Splitter Insertion for AQFP Circuits," 2022 Design Automation Conference (DAC).

Scalable and Generic Logic Synthesis **02/2020 — 03/2024**
Advisor: Prof. Giovanni De Micheli *Integrated Systems Lab, EPFL*
Collaborators: Dr. Alan Mishchenko (UC Berkeley), Dr. Heinz Riener (Cadence, Germany)

- **Siang-Yun Lee**, Heinz Riener, Alan Mishchenko, Robert K. Brayton, and Giovanni De Micheli. "A Simulation-Guided Paradigm for Logic Synthesis and Verification," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD). 2022.
- **Siang-Yun Lee** and Giovanni De Micheli. "Heuristic Logic Resynthesis Algorithms at the Core of Peephole Optimization," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD). 2023.

Threshold Logic Canonicalization and Weight-Sharing Synthesis **08/2016 — 08/2019**
Advisor: Prof. Jie-Hong Roland Jiang *Bachelor Project, NTU*

- **Siang-Yun Lee**, Nian-Ze Lee, and Jie-Hong R. Jiang. "Searching Parallel Separating Hyperplanes for Effective Compression of Threshold Logic Networks," 2019 International Conference on Computer-Aided Design (ICCAD).
- **Siang-Yun Lee**, Nian-Ze Lee, and Jie-Hong R. Jiang. "Canonicalization of Threshold Logic Representation and Its Applications," 2018 International Conference on Computer-Aided Design (ICCAD).

Teaching Activities

Lecturer	CS-724: Advanced Logic Synthesis and Quantum Computing	Spring 2022, 2023 (EPFL)
Ph.D. assistant	CS-472: Design Technologies for Integrated Systems	Fall 2020, 2021, 2022 (EPFL)
Ph.D. assistant	CS-173: Digital System Design	Spring 2020, 2021, 2022 (EPFL)
Course design & assistant	Cornerstone EECS Design and Implementation	Spring 2018, 2019 (NTU)
Student assistant	Switching Circuit and Logic Design	Fall 2016 (NTU)
Course design & instructor	Interactive hands-on camp introducing EE to high school students	2018 — 2019 (TimeMap)

Awards & Grants

- **Best Student Paper Award**, International Workshop on Logic and Synthesis (IWLS), 07/2022.
  — For the paper "An Automated Testing and Debugging Toolkit for Gate-Level Logic Synthesis Applications"
- **First Prize Award**, IWLS Programming Contest, 07/2022.
- **First Prize Award**, ACM/SIGDA CADathlon Programming Contest (ICCAD, USA), 11/2019.
- EPFL Doctoral Program in Computer and Communication Sciences (EDIC) Fellowship, 2019.
- **First Prize Award**, Integrated Circuits Computer Aided Design Contest (Ministry of Education, Taiwan), 11/2017.
  — For the project "Input Sequence Generator for System Verilog Assertion Checking"
- Undergraduate Student Research Program, Ministry of Science and Technology, Taiwan, 2018.
- Undergraduate Scholarship, TSMC–NTU Joint Research Center, 2017.
- **Fourth Award** in cellular and molecular biology, Intel International Science and Engineering Fair (ISEF, USA), 05/2015.
  — For the project "Mitochondrial Protein CYP11A1 Changes Mitochondrial Morphology"

Skills

Computer Languages	C/C++, Python, Verilog, VHDL, JavaScript
Software Tools	Git, LaTeX, EDA tools (Synopsys Design Compiler, Intel Quartus II, ModelSim, etc.), Matlab
Human Languages	English (fluent), Mandarin (native), French (A2), German (B1)