EPFL

Thèse n° 9278

Practical Compilation of Quantum Programs

Présentée le 12 août 2022

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

Bruno SCHMITT ANTUNES

Acceptée sur proposition du jury

Prof. P. Ienne, président du jury
Prof. G. De Micheli, Dr M. Soeken, directeurs de thèse
Prof. F. T. Chong, rapporteur
Prof. R. Wille, rapporteur
Prof. E. Charbon, rapporteur

École
polytechnique
fédérale
de Lausanne

2022

To Fernando, Felipe, Gilda Helena and Francisco Olinto.

Acknowledgements

First, I wish to thank my advisor, Prof. Giovanni De Micheli, for accepting me into his research lab and providing guidance and support throughout my time at EPFL. I want to extend my most profound appreciation to my co-advisor, Dr. Mathias Soeken, whom I now consider a friend, for bringing me to EPFL, introducing me to the field of quantum computing, and for all the valuable ideas and discussions. This thesis would not have been possible without his help.

I also wish to thank my jury members Prof. Paolo Ienne, Prof. Edoardo Charbon, Prof. Frederic T. Chong, and Prof. Robert Wille, for judging the thesis and their helpful comments and suggestions.

I'm grateful to Dr. Martin Roetteler, Dr. Stephen Jordan from Microsoft, and Dr. Ali Javadi-Abhari from IBM for the opportunities to work on exciting summer internships. I'm also thankful to all colleagues I interacted with during these internships.

Throughout my years as a graduate student, I have worked alongside talented researchers and fellow students who have helped me in my research. For that, I wish to thank Alan Mishchenko, Heinz Riener, Siang-Yun (Sonia) Lee, Eleonora Testa, Giulia Meuli, Kaitlin Smith, Winston Haaswijk, Fereshte Mozafari, and all other LSI members.

My time in Lausanne would have been far less joyful if not for the friends I made there. I thank Delio, Fabian, Luis Henrique, Manuella, Jason, and Rabei. A big *obrigado* to my family and friends in Brazil, Netherlands, and Italy— especially Carmen Schmitt, Bruna Lacerda, Fabricio Pavan, Virgilio Peixoto, and Daniel Campelo for visiting me. I'm also grateful to my cousin Guilherme, his wife Juliana, and their son Noah for bringing great happiness to my weekends when they moved there: Thanks for all the family lunches.

Finally, I thank my parents and brothers, to whom I dedicate this thesis, for their immeasurable support during this journey. They are the inspiration that keeps me aiming at higher goals in my life.

Lausanne, June 16, 2022 Bruno Schmitt

Abstract

It's been a little more than 40 years since researchers first suggested exploiting quantum physics to build more powerful computers. During this time, we have seen the development of many quantum algorithms and significant technological advances to make these devices. As a result, at this point, large-scale quantum computers, capable of providing valuable solutions to complex problems, seem like a certainty, even if still distant.

Nonetheless, there is a great gap between the communities of quantum algorithms and quantum devices researchers. On the one hand, algorithm designers most often describe algorithms in a high level of abstraction, using a mixture of natural language, pseudocode, and mathematical formulas—a form deriving asymptotic complexity estimates while shielding researchers from the low-level complexities and restrictions. On the other hand, physical devices only understand algorithms implemented using the primitive low-level abstractions they support.

Most programming systems available for quantum computing are intertwined with the quantum circuit model, so developers must implement algorithms in terms of low-level unitary operators. Not surprisingly, the implementation of quantum algorithms on such a low level of abstraction is very time-consuming, error-prone, and results in non-portable programs—given the technological diversity of quantum devices.

In this thesis, I study problems related to the compilation of quantum programs, seeking forms of augmenting the expressive power of current frameworks and narrowing the gap between algorithmic research and concrete implementations. I focus on scalability and practicality—in particular, together with theoretical investigations, I developed concrete algorithms that are performant and scalable. The embodiment of my research contribution is a compiler companion library for the synthesis and compilation of quantum circuits called `tweedledum`.

Keywords: design automation, logic synthesis, reversible logic, compilation, quantum computing, quantum programming.

Zusammenfassung

Es ist etwas mehr als 40 Jahre her, dass Wissenschaftler zum ersten Mal Quantenphysik als Grundlage für leistungsstarke Computer vorgeschlagen haben. Seitdem beobachten wir die Entwicklung vieler Quantenalgorithmen sowie signifikante technologische Durchbrüche. All dies führt heutzutage zu einer hohen Zuversicht, dass leistungsstarke Quantencomputer in der Zukunft–wenn auch weit entfernt–wertvolle Lösungen zu komplexen Problemen liefern werden.

Nichtsdestotrotz existiert eine große Abstraktionsdifferenz zwischen Wissenschaftlern im Bereich von Quantenalgorithmen und denen, die an physikalischen Quantencomputern arbeiten. Entwickler von Quantenalgorithmen beschreiben jene mit einer Mischung aus natürlicher Sprache, Pseudocode, sowie mathematischen Formeln oft auf einer sehr hohen Abstraktionsebene. Dabei leiten sie asymptotische Komplexitätsergebnisse her, ohne auf die Implementierungsdetails unterer Abstraktionsebenen einzugehen, die von konkreten physikalischen Quantencomputern verstanden werden.

Die meisten Programmiersysteme für Quantencomputer sind stark mit dem Quantenschaltkreis Modell verflochten. Daher müssen Programmierer Algorithmen auf Basis von maschinennahen unitären Operationen implementieren. Es überrascht nicht, dass dies sehr zeitintensiv und fehleranfällig ist. Zudem lassen sich die Programme nicht einfach portieren, da verschiedene Quantencomputer sehr unterschiedliche Fähigkeiten haben.

In dieser Dissertation untersuche ich Compiler von Quantenprogrammen. Diese erweitern existierende Programmiersysteme durch Ausdrucksmöglichkeiten auf höheren Abstraktionsebenen und führen schlussendlich dazu, dass die Abstraktionsdifferenz zwischen algorithmischen Beschreibungen und konkreten Implementierungen verringert wird. Den Fokus lege ich auf Erweiterbarkeit, Performanz, und praktische Anwendbarkeit. Basierend auf theoretischen Untersuchungen habe ich neue performante und universelle Algorithmen entwickelt. Die Softwarebibliothek `tweedledum` fasst meine Arbeit zusammen und erlaubt die Integration meiner wissenschaftlichen Beiträge in existierende Programmiersysteme für Quantencomputer.

Schlüsselwörter: Entwurfsautomatisierung, Logiksynthese, reversible Logik, Compilation, Quantencomputer, Programmierung von Quantencomputern

Contents

Acknowledgements i

Abstract (English/Deutsch) iii

Introduction 1
- Contributions . . . 2
- Outline . . . 4

I Background 5

1 Logic synthesis 7
- 1.1 Boolean functions . . . 7
- 1.2 Logic data structures . . . 8
  - 1.2.1 Truth table . . . 8
  - 1.2.2 Two-level expressions . . . 9
  - 1.2.3 Binary decision diagrams . . . 9
  - 1.2.4 Multi-level logic networks . . . 11
- 1.3 Boolean satisfiability . . . 14

2 Quantum computing 15
- 2.1 Quantum bits . . . 15
- 2.2 Quantum operators . . . 16
- 2.3 The circuit model . . . 18
- 2.4 Representation of quantum functionality . . . 19
- 2.5 Oracles . . . 20

II Synthesis 23

3 Introduction to quantum circuits synthesis 25
- 3.1 Synthesis problems . . . 26

4 ESOP-based synthesis **29**
4.1 Motivation . . . 29
4.2 Contributions . . . 29
4.3 Previous work . . . 30
4.4 Divide-and-conquer extraction . . . 34
4.4.1 Selection heuristics . . . 35
4.5 Experimental results . . . 35
4.6 Use case: ESOP-based reversible logic synthesis . . . 38
4.6.1 Experimental results . . . 39
4.7 Summary . . . 40

5 XAG-based synthesis **43**
5.1 Motivation . . . 43
5.2 Contributions . . . 44
5.3 Previous work . . . 44
5.4 Synthesis flow . . . 46
5.5 Experimental results . . . 51
5.6 Summary . . . 52

6 Symbolic algorithms for permutation synthesis **55**
6.1 Motivation . . . 55
6.2 Technical background . . . 56
6.2.1 Graphs . . . 56
6.2.2 Permutation . . . 57
6.2.3 Reconfiguration problems . . . 57
6.2.4 State space . . . 58
6.2.5 Decision diagrams . . . 58
6.3 Contributions . . . 59
6.4 A*-based algorithm . . . 59
6.5 SAT-based algorithm . . . 61
6.6 πDD-based algorithm . . . 63
6.7 Experimental results . . . 64
6.8 Summary . . . 67

7 SAT-based linear synthesis **69**
7.1 Motivation . . . 69
7.2 Technical background . . . 69
7.3 Previous work . . . 71
7.4 Contributions . . . 72
7.5 Synthesis algorithm . . . 72
7.6 Discussion . . . 74

III Compilation **75**

8 Introduction to compilation of quantum programs **77**
8.1 Contributions 79

9 The `tweedledum` library **81**
9.1 The intermediate representation 81
9.1.1 Fundamental concepts 81
9.2 Synthesis 83
9.3 Compilation 85
9.3.1 Utility 85
9.3.2 Decomposition 86
9.3.3 Mapping 87
9.3.4 Optimization 91
9.4 Show case: Boolean function compilation 93
9.4.1 IBM's challenge: The Zed city problem 97
9.5 Show case: Mapping 99
9.6 Summary 101

10 Conclusion **103**
10.1 Future directions 104

Bibliography **107**

Curriculum Vitae **123**

Introduction

The history of quantum computing dates back to the early 1980s when Paul Benioff demonstrated a quantum mechanical model of a Turing machine [23]. Around the same time, Yuri Manin and Richard Feynman argued about the inherently exponential cost of simulating generic quantum systems using conventional classical digital computers [89, 60]. Feynman suggested, in his work, that a computer that uses quantum mechanical phenomena for computation might simulate quantum systems more efficiently. Since then, a series of papers [53, 39, 27, 8] have formalized an abstract model of quantum computation and the concept of a quantum Turing machine, and many researchers have striven to build such devices.

Technology advances have motivated a new wave of investment in quantum research in the past few years. Today we are in the Noise Intermediate-Scale Quantum (NISQ, [110]) era, characterized by the appearance of quantum computers with sizes ranging from fifty to a few hundred qubits—e.g., IBM recently announced a 127-qubit device [3]. At this point, large-scale quantum computers, capable of providing valuable solutions to complex problems, seem like a certainty, even if still distant. As a result, we have seen a rapid increase in research to create new quantum algorithms and research to develop design tools for quantum computers. I refer to the "Quantum Algorithm Zoo" website for algorithmic advances [76]; It holds a comprehensive list of quantum algorithms with their respective speed-up factors. On the design tool front, we have seen the appearance of a large variety of programming environments—for instance `Q#` [145], `Silq` [28], `Quipper` [68], `Qiskit` [150], `Cirq` [55] `PyQuil/Forest` [131], `Pennylane` [26], `ProjectQ` [142], `StrawberryFields` [78]. Such frameworks aim to bridge the gap between algorithm researchers and developers by providing ways of implementing quantum algorithms using a programming language or API, which a compiler translates into an executable format for a quantum device.

Nonetheless, the gap between these communities is still significant. Quantum algorithm designers often work at a high level of abstraction and do not consider current frameworks and hardware capabilities a limiting factor when conceiving and describing new algorithms [41]. On the other hand, most programming systems available for quantum computing are intertwined with the quantum circuit model, so developers must implement algorithms in terms of low-level unitary operators. Not surprisingly, the implementation of quantum algorithms on such a low level of abstraction is very time-consuming, error-prone, and results in non-portable programs—given the technological

diversity of quantum devices.

Besides being essential for executing programs on physical devices, designs tools also play an indispensable role in resource estimation, i.e., estimating resources required to implement quantum algorithms on future fault-tolerant hardware. There is a whole field of resource estimation that is motivated by the need to understand better the power of quantum computing to make policy decisions, stimulate investment, and guide research towards technological advances that will lead to scalable devices.

In this thesis, I study problems related to the compilation of quantum programs, seeking forms of augmenting the expressive power of current frameworks and narrowing the gap between algorithmic research and concrete implementations. I focus on scalability and practicality—in particular, alongside theoretical investigations, I developed concrete algorithms that are performant and scalable. The embodiment of my research contribution is a compiler companion library for the synthesis and compilation of quantum circuits called `tweedledum`. In contrast to most solutions, I designed it to enhance other compilers and frameworks, and some of these tools indeed use it already. The following section briefly outlines the contributions.

CONTRIBUTIONS

Many quantum algorithms use operations with classically described behavior, particularly oracular ones [70, 40, 88, 118, 143, 46]. An algorithm uses an oracle to access a Boolean function, i.e., it queries the oracle on some input to obtain the corresponding function's output.

Researchers use oracles extensively when studying quantum algorithms' complexity because counting the number of queries required to evaluate a function is more straightforward than counting the number of computational steps. Thus, to try inferring nontrivial lower bounds more readily, investigators characterize the computational complexity by the asymptotic growth rate of the number of queries with growing input size—an analysis that only assumes the existence of an oracle.

However, a developer needs to provide a concrete implementation of the whole algorithm, oracles included, to execute it on an actual device or accurately estimate the resources necessary to perform such algorithms. Such an implementation must consist of a sequence of elementary quantum operators supported by the underlying hardware. Furthermore, due to the physical properties of quantum states, all quantum operations need to be reversible; thus, a classical function that defines the oracle's behavior must be reversible, a characteristic not often found in real-world problems. On the contrary, we often employ quantum oracles defined by complex, irreversible functions.

Using most of today's frameworks, we have an impractical situation. We are demanding developers to reversibly implement the desired function using low-level quantum operations while attempting to minimize quantum resources usage, i.e., the number of qubits and operations.

This thesis aims at remedying this situation by allowing designers to implement the

complicated classical subroutines on a high level of abstraction and then automatically translate such implementations into low-level quantum circuits. We present various contributions that support our goal. The first tackles the problem of ESOP-based synthesis scalability [137], a technique capable of generating reversible circuits with an optimal number of qubits due to a natural affinity between exclusive sum-of-products (ESOP) expressions and Toffoli gates. We describe a new algorithm that outperforms state-of-the-art methods in both scalability and effectiveness. This improvement also benefits some hierarchical synthesis techniques, specifically those relying on direct synthesis to generate circuits for small logic functions decomposed from a large one, e.g., LUT-based hierarchical reversible logic synthesis (LHRS) [138]. It enables those methods to split a logic function into fewer pieces, which, in turn, can minimize the number of required qubits.

The second contribution is an improvement to a hierarchical synthesis algorithm specifically designed to work on xor-and inverter graphs (XAG). This technique synthesizes circuits over a fault-tolerant gate set, Clifford+T, and focuses on minimizing the total number of gates and, more specifically, the number of T gates, which are more expensive to error correct [11, 63]. The improved technique can significantly reduce the number of qubits and Clifford gates. Crucially, these improvements are possible without increasing the number of T gates or the execution time.

The thesis also presents the results of research directed towards the development and analysis of symbolic algorithms for solving two reconfiguration problems [12, 155] that appear in the context of quantum circuit mapping, which is the process of modifying a high-level circuit that assumes full qubit connectivity, into a lower-level one that respects the device's connectivity (or coupling) constraints. This process is an essential step during the compilation and frequently requires the inclusion of additional operations, in this case, SWAP operations. We can use the three proposed algorithms to do the mapping itself or as a post-mapping optimization technique. All three algorithms guarantee local optimality, i.e., they generate permutation circuits with the optimal number of SWAPs or optimal depth.

While performant, SWAP-based mappers and the synthesis of permutation circuits using SWAPs are suboptimal. The reason is that today's quantum hardware cannot execute SWAP gates directly. Thus, we must translate SWAPs into a sequence of CNOT gates during compilation. In various applications, the cost of these subcircuits dominates the total costs. Fortunately, we can improve the implementation of permutation circuits by leveraging the properties of the CNOT gate. The last synthesis technique we present in this thesis is the SAT-based method for generating linear reversible circuits [108], i.e., those that are composed only of CNOT gates. Although this technique is much more general, we mainly employ it to synthesize permutation circuits and disprove a 15-year-old conjecture that reversal is at least as depth-intensive to synthesize with CNOTs as any other permutation on a path [83].

Finally, we present `tweedledum`, an extensible and efficient open-source library for quantum circuit synthesis and compilation. It has state-of-the-art performance in many

compilation tasks relevant to NISQ and fault-tolerant applications. We describe most of its state-of-the-art techniques for synthesizing and compiling quantum circuits, alongside practical implementation improvements to some of them, and the library's core: an intuitive and flexible intermediate representation that supports different abstraction levels across the same circuit structure.

The contributions above are published in [123, 124, 120, 119, 37]. In addition to the work described here, other publications completed during my doctoral studies include [136, 135, 93, 130, 113, 134, 122].

OUTLINE

This thesis is divided into three parts:

- Part I (Chapters 1 and 2) covers background material most of necessary for this thesis. Chapter 1 introduces concepts from the field of classical logic synthesis and some of the various ways employed to represent Boolean functions. Chapter 2 provides a brief introduction to quantum computing.

- Part II (Chapters 3 to 7) covers synthesis algorithms. In Chapter 3, we introduce quantum circuit synthesis. Chapters 4 and 5 tackle the problem of synthesizing large, complex, irreversible Boolean functions into quantum circuits. Chapter 6 studies two problem on graphs that appear in the context of quantum circuit mapping and provide three SWAP-based algorithms to synthesize permutation circuits. Lastly, Chapter 7 deals with the optimal synthesis of linear circuits.

- Part III (Chapters 8 and 9). Chapter 8 briefly introduces the whole compilation process for quantum programs. In Chapter 9 we describe `tweedledum`—an compiler companion library for the synthesis and compilation of quantum circuits that embodies our research. We demonstrate its power using two show case examples: oracle synthesis and quantum circuit mapping.

Part I

Background

Chapter 1

Logic synthesis

As general notation, we are using $[n] = \{1, \ldots, n\}$.

1.1. BOOLEAN FUNCTIONS

A *Boolean variable* x is a variable that takes one of the two values from the domain $\{\text{false}, \text{true}\}$, or $\{0, 1\}$. Variables are often denoted by $x_1, x_2, x_3, \ldots$; but we will often find it convenient to use just the numerals $1, 2, 3, \ldots$ to stand for variables, or write just j instead of x_j, because it takes less space. A *positive literal* is the Boolean variable x and a *negative literal* is its complement $\bar{x}$. Similarly to variables, we can write '2' to denote the 'x_2' literal, and '$\bar{2}$' for its complement. The Boolean AND of k literals is a *cube*, or product, i.e., $c = l_1 \wedge \cdots \wedge l_k$ (we may omit the symbol $\wedge$ in forming cubes, e.g., $l_1 \wedge \cdots \wedge l_k = l_1 \cdots l_k$). If a variable is not represented by a positive literal or a negative literal in a cube, then its value is said to be a *don't care literal.* A *minterm* is a cube, in which every variable is represented by either a negative or positive literal. A cube with k don't care literal values covers 2^k minterms.

A *multiple-output Boolean function* $f : \{0,1\}^n \to \{0,1\}^m$ maps n Boolean input variables to m Boolean output values. We refer to

$$\mathbb{B}_{n,m} = \{f \mid \{0,1\}^n \to \{0,1\}^m\} \tag{1.1}$$

as the set of all multiple-output Boolean functions with n inputs and m outputs, where $m, n \geq 1$. We write $\mathbb{B}_n = \mathbb{B}_{n,1}$ to denote the set of all single-output Boolean functions. We can represent f as a m-tuple of n-variable Boolean functions $(f_1, \ldots, f_m)$ where $f_i \in \mathbb{B}_n$ for each $i \in [m]$ and thus $f(x) = (f_1(x), \ldots, f_m(x))$ for each $x \in \{0,1\}^n$. We use $y_1, \ldots, y_{m-1}$ to denote the outputs of a function.

The *support* of f is the subset of variables that influence the output value of the function f. Unless stated otherwise, we assume that a Boolean function is completely specified. The *cofactors* are derived from the function by substituting constant values for the input variables. For example, Boole's expansion of a function f, often called Shannon's expansion, where $f_{\bar{x}_i} = f(x_i = 0)$ and $f_{x_i} = f(x_i = 1)$ are the negative and

positive cofactors of the function f with respect to variable x_i, respectively.

1.2. LOGIC DATA STRUCTURES

This section introduces the data structures used in this thesis for logic synthesis and optimization. These include truth tables, two-level expressions, binary decision diagrams, and multi-level logic networks.

1.2.1 Truth table

A natural way of representing a Boolean function is through the exhaustive enumeration of its mapping, thus creating a table in which every truth assignment of its input variables has a corresponding function value listed. A *truth table* is precisely that.

Example 1.1. *The function* $\text{prime}_3(x)$ *takes a 3-bit number and returns true if this number is prime, i.e.,* $\text{prime}_3(x) = [(x_3x_2x_1)_2$ *is prime*$]$. *We can represent it using the following truth table:*

x_3	x_2	x_1	prime_3
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Note that the input side of a truth table is redundant, e.g., given two functions with the same number of inputs, their truth table will only differ on the output column. Hence, we can represent the truth table for a Boolean function f as a bitstring $b_{2^n-1} \dots b_1 b_0$ where $b_x = f(x)$ when $x = (x_n \dots x_1)_2$. In other words, each character in the bitstring corresponds to the function value of f evaluated at some assignment to its variables. The most-significant bit b_{2^n-1} corresponds to $f(1, \dots, 1)$ and the least-significant bit b_0 corresponds $f(0, \dots, 0)$. Thus the truth table representation for the $\text{prime}_3(x)$ function in Example 1.1 is 10101100, or `0xAC` in hexadecimal.

Unfortunately, truth tables are impractical to represent large functions because their size grows exponentially with the number of input bits, 2^n. Nonetheless, they are tremendously useful to represent and manipulate small functions. For functions with up to 16 variables, truth tables are typically much faster than alternative representations. Furthermore, truth tables are *canonical* representations of Boolean functions: Two Boolean functions are equivalent if and only if they have the same truth table. Canonicity is a valuable property in many applications.

1.2.2 Two-level expressions

We can also use Boolean expressions composed of literals, an inner operator, and an outer operator to represent Boolean functions. For example, the *sum-of-products* (or SOP) representation has '$\wedge$' (logical AND) as inner operator and '$\vee$' (logical OR) as outer operator. An SOP for an n-variable Boolean function has the form:

$$f(x_n, \ldots, x_1) = \bigvee_{i=1}^{m} (x_n^{p_{n,i}} \wedge \cdots \wedge x_1^{p_{1,i}}) \tag{1.2}$$

for some m and polarities $p_{j,i} \in [3]$, meaning $x_j^0 = 1$ (a *don't care*), $x_j^1 = \bar{x}_j$, and $x_j^2 = x_j$. Note that each term corresponds to a cube, and their order does not change the function. An SOP evaluates to true if *at least one* term is true. An SOP in which all terms are minterms and unique (i.e., without repetition) is canonical for a function up to the terms' order. However, such SOPs are not of much practical interest. In general, we are interested in finding the most concise SOP-form for a Boolean function. Such forms are not canonical, and finding them is an intractable problem (in fact, NP-complete []).

Example 1.2. *The function* $\text{prime}_4(x)$ *takes a 4-bit number and returns true if this number is prime, i.e.,* $\text{prime}_4(x) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$. *We can represent it using the following SOP:*

$$\begin{aligned}\text{prime}_4(x) = {} & \bar{x}_4\bar{x}_3x_2\bar{x}_1 \vee \bar{x}_4\bar{x}_3x_2x_1 \vee \bar{x}_4x_3\bar{x}_2x_1 \\ & \vee \bar{x}_4x_3x_2x_1 \vee x_4\bar{x}_3x_2x_1 \vee x_4x_3\bar{x}_2x_1\end{aligned}$$

or, more concisely:

$$\text{prime}_4(x) = x_3\bar{x}_2x_1 \vee \bar{x}_3x_2x_1 \vee \bar{x}_4x_3x_1 \vee \bar{x}_4\bar{x}_3x_2.$$

In quantum computing, implementing logical exclusive-OR (XOR) is cheaper than implementing logical OR. Hence, the use of two-level forms focuses on a crucial variation of (1.2) that emerges when changing the outer operator '$\vee$' to '$\oplus$'. Such an expression is called *exclusive-or sum-of-products* (ESOP).

Example 1.3. *The function* $\text{prime}_4(x)$ *takes a 4-bit number and returns true if this number is prime, i.e.,* $\text{prime}_4(x) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$. *We can represent it using the following ESOP:*

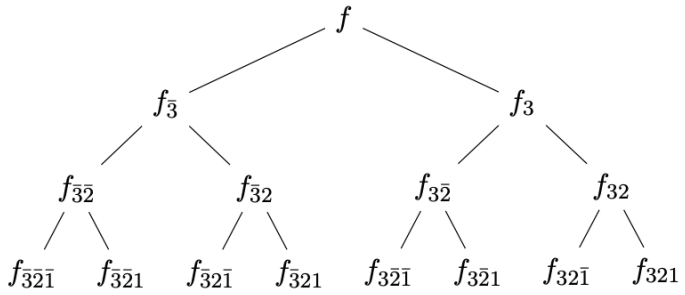
$$\text{prime}_4(x) = x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2.$$

1.2.3 Binary decision diagrams

First proposed by Lee [85] and further developed by Akers [9], a *binary decision diagram* (or BDD for short) is a graph-based representation of a Boolean function. In their original form, BDDs are not canonical. To canonicalize the representation, Bryant [34, 35] introduced constraints on variable ordering and proposed several reduction rules, leading

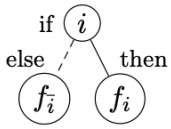
to the well-known *reduced ordered BDD*. For the remainder of this thesis, we use "BDD" to denote a binary decision diagram that is ordered and reduced.

The basic idea behind BDDs is a divide-and-conquer scheme based on Boole's expansion theorem. Recall that this theorem allows us to rewrite any Boolean f as $x_i f_{x_i} \vee \bar{x}_i f_{\bar{x}_i}$—effectively dividing f into two subfunctions, or cofactors, f_{x_i} and $f_{\bar{x}_i}$, which are combined with different factors. We can depict the recursive application of this theorem to its limits (i.e., while the resulting subfunctions have input variables) as a tree, i.e.,



(1.3)

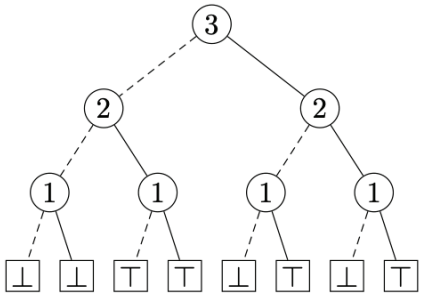
Here we decided to omit the factors and use numerals to denote literals. The leaves of this tree, i.e., the functions at the bottom layer, are either the constants zero (false) or one (true). Also, note that we can interpret Boole's expansion $f = x_i f_{x_i} \vee \bar{x}_i f_{\bar{x}_i}$ as a if-then-else, with x_i being the condition and the cofactors being the branches. Visually, we can represent it as



(1.4)

A binary decision tree is a structure based on both (1.3) and (1.4). The following example illustrates it.

Example 1.4. *A binary decision* ***tree*** *for* $\mathrm{prime}_3(x)$*:*

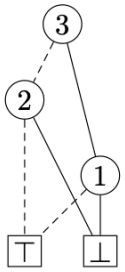


A binary decision tree consists of several decision vertices and terminal vertices. Terminal vertices are labeled with a Boolean value, either true ($\boxed{\top}$) or false ($\boxed{\bot}$). Decision vertices ($\textcircled{i}$) are labeled with a name or index designating a Boolean variable x_i and have two successors called `LO` and `HI`. The `LO` successor is drawn as a dashed line and corresponds to the node's decision variable being equal to zero. The `HI` successor, drawn as a solid line, corresponds to the node's decision variable being equal to one. Using a binary

decision tree, one can determine the value of a Boolean function for any given variable assignment by following a path from the root vertex to a terminal vertex.

Note that each vertex in a binary decision tree can be associated with a Boolean function—which are subfunctions of the root vertex's one. Binary decision diagrams use the fact that these subfunctions repeatedly occur when representing functions of practical interest and thus need to be represented only once. We guarantee their unique representation by applying the reductions rules described in [34].

Example 1.5. *The binary decision* ***diagram*** *for* $\mathrm{prime}_3(x)$*:*



Binary decision diagrams are ordered in the sense that the Boole's decomposition is applied with respect to some given variable ordering which also has an effect on the is number of nodes. Improving the variable ordering for BDDs is NP-complete [30] and many heuristics have been presented that aim at finding a good ordering.

1.2.4 Multi-level logic networks

A logic network for functions of n variables $(x_1, \ldots, x_n)$ is a sequence of r gates $x_n, \ldots, x_{n+r}$ with the property that each gates combines at least two of the preceding gates:

$$x_i = f_i(x_{i_0} \oplus p_{i_0}, \ldots, x_{i_{k_i}} \oplus p_{i_{k_i}}), \qquad \text{for } n < i \leq n + r. \tag{1.5}$$

A k_i-input Boolean function f_i defines the behavior of each gate. The fan-ins $x_{i_j} \oplus p_{i_j}$, where $0 \leq i_j < i$, are either the constant $x_0 = 0$, primary inputs, or preceding gates. The polarity flags p_{i_j} are Boolean constants used to complement the gate's fan-ins. (Note that for this general definition, these polarity flags are unnecessary as the negation of a fan-in can be incorporated into the gate function f_i. Nevertheless, we opted to leave them explicit.) We refer to the set $\{x_1, x_2, \ldots, x_{n+r}\}$ as nodes of the logic network. A Boolean function described by the logic network is represented in terms of outputs $y_1, \ldots, y_m$, where

$$y_j = x_{o_j} \oplus p_j \tag{1.6}$$

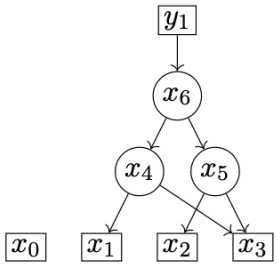
and $0 \leq o_j \leq n + r$. We use y_j to refer to the output of a preceding node x_{o_j} with polarity p_j.

We can represent logic networks as directed acyclic graphs (DAG) in which the vertices correspond to the constant-0 input, the n primary inputs, the r gates, and the m primary outputs. The arcs (directed edges) represent inputs and can only point to

previous vertices, i.e., there is an arc $x_j \longrightarrow x_i$ if $j > i > n$ and x_i belongs to x_j's set of fan-ins. Also, if a gate x_{o_j} is an output, then there is an arc $y_j \longrightarrow x_{o_j}$—we say that x_{o_j} drives output y_j.

Example 1.6. *We can implement a logic network for* $\text{prime}_3(x)$ *as follows:*

$$x_4 = x_3 \wedge x_1$$
$$x_5 = \bar{x}_3 \wedge x_2$$
$$x_6 = x_4 \vee x_5$$



The output is $y_0 = x_6$*. We could also implement it using a single gate* $x_4 = \bar{x}_3 x_2 \vee x_3 x_1$ *that is also the output* y_1*.*

Observe that our definition of a logic network does not impose any restriction on gates' functionality—f_i in (1.5). Hence, when representing it as a DAG, each vertex corresponds to a logic function with an unbounded number of inputs. Interestingly, we can represent these functions as truth tables or two-level expressions. In the latter case, we can leverage the proven efficiency of two-level optimization techniques to optimize these functions. However, while potentially very compact, such a DAG does not support robust logic optimization because it imposes impractical requirements on optimization techniques, e.g., the ability to deal with all functions' types and sizes. Furthermore, the cumulative memory footprint for each functionally unbounded vertex is potentially huge.

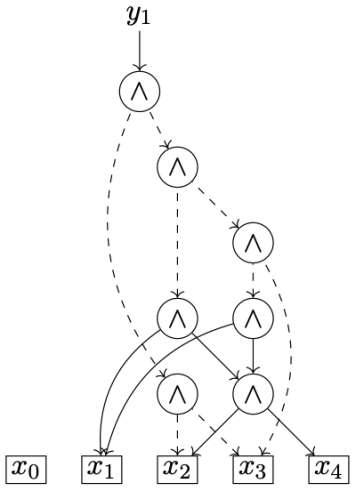
This definition, therefore, is of little practical interest due to excessive generality. We can specialize the definition by constraining the internal vertices' functionality to alleviate this issue. In extreme cases, one can use the same function, with a bounded number of inputs, for all vertices and add polarity attributes to the arcs. In principle, this restriction increases the representation size in the number of vertices and arcs; in practice, it unlocks better (smaller) representations because it supports more effective logic optimization techniques.

Before looking at concrete specialized logic networks, let me introduce some properties. A logic network is called k-regular, if $k_i = k$ for all $n < i \leq n + r$, i.e., all gates have the same number of fan-ins. It is called k-feasible, if $k_i \leq k$ for all vertices. We say a logic network is *homogeneous*, if $f_i = f_j$ for all $n < i < j \leq n + r$.

This thesis uses three common specializations of general definition:

And-Inverter graph (AIG) [82]. An AIG is a 2-regular homogeneous logic network, in which each gate function f_i is the 2-input AND. Thus, the network is composed of AND gates and inverters, which are modelled using the polarity flags—hence our decision of expressing them explicitly on the general definition. Due to their regularity and homogeneity, AIGs are particularly efficient for Boolean function representation and reasoning.

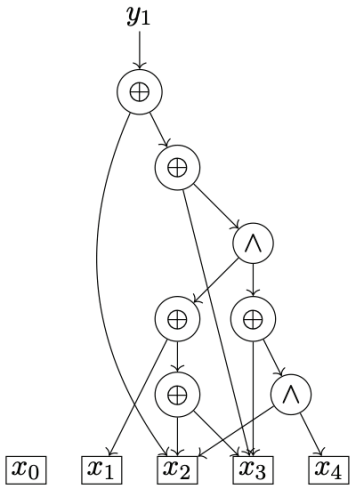
Example 1.7. *We can implement an AIG for* $\text{prime}_4(x)$ *as follows:*



Xor-And graph (XAG). An XAG is a 2-regular logic network that is not homogeneous since its gates can be either 2-input AND or XOR. Similarly to AIGs, inverters are modelled using polarity flags. Thus each gate has one of following forms:

$$x_i = x_{i_0} \oplus x_{i_1} \quad \text{or} \quad x_i = (x_{i_0} \oplus p_{i_0}) \wedge (x_{i_1} \oplus p_{i_1}), \tag{1.7}$$

Example 1.8. *We can implement an XAG for* $\text{prime}_4(x)$ *as follows:*

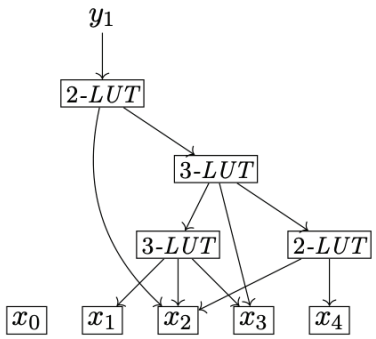


An interesting characteristic of AIGs and XAGs is that their simple data structure enables quick and cheap structural hashing among nodes: Two nodes with the same inputs under the same polarity conditions are merged (similar to a reduction rule of BDD). However, unlike BDDs, they are not canonical, even when structurally hashed.

k-LUT network. A k-input lookup table (k-LUT) is a hardware device that can implement any Boolean function up to k inputs and thus can be modeled as a truth table.

A k-LUT network is a k-feasible logic network in which the gates' functionality is represented by a truth table with at most k inputs. Since a truth table can represent any Boolean function, there is no need for polarity flags, i.e., $p_{i,j} = 0$ for all gate fan-ins and $p_j = 0$ for all outputs.

Example 1.9. *We can implement a k-LUT network, where $k = 3$, for $\text{prime}_4(x)$ as follows:*



It is worth noting that we rarely implement functions as k-LUT networks. Their original use was to map logic designs into the building blocks of field-programmable gate arrays (FPGA) [44], which can compute any Boolean function up to a given number of inputs, i.e., look-up tables (LUT). Later, LUT mappers found a successful application in logic synthesis and circuits optimization [102] because of their ability to "cut" (decompose) large Boolean functions into smaller ones. There are several efficient state-of-the-art mappers [38, 116, 111] and post mapping optimization techniques [121, 87] available. Traditionally, they aim to minimize delay and area of the resulting circuit.

1.3. BOOLEAN SATISFIABILITY

The Boolean satisfiability problem (hereafter SAT) asks whether a given propositional formula representing an n-variable Boolean function f is satisfiable or not, i.e., whether there exists an assignment of variables $x \in \mathbb{B}^n$ such that $f(x) = 1$. When such an assignment does not exist, the formula is said to be unsatisfiable (UNSAT).

The SAT problem is NP-complete [45]. Still, several instances of practical interest are efficiently solvable using state-of-the-art SAT solvers [81, 29, 51]—even instances containing tens of thousands of variables and millions of constraints. Most modern SAT solvers require a conjunctive normal form (CNF) encoding of the problem. In such encoding, a given property's presence (absence) is represented by a positive (negative) literal of a variable. The combined literals form clauses—i.e., a disjunction of literals. The conjunction of clauses forms the CNF. The encoding of a problem is crucial and can significantly impact the execution time of an SAT solver.

Another important characteristic of modern solvers is that they can accept a set of assumptions, which enforces a value to a variable. The process of determining the satisfiability of a problem under given assumptions, is called incremental SAT solving [59]. For more details on Boolean satisfiability, see [29, 81].

Chapter 2

Quantum computing

Quantum computing is the study of performing computational tasks using a quantum mechanical system, e.g., a quantum computer. Therefore, the subject is often mistakenly thought of as complex, requiring years of physics training and understanding all about the wave-particle duality, boson-fermion statistics, or even Schrödinger's equation. Fortunately, this thinking could not be further from the truth: understanding almost the entire body of research on quantum information and computing requires only knowing how to manipulate vectors and matrices whose entries are complex numbers [4].

The core of quantum computation consists of encoding information on the state of a quantum system. We represent such a state as a normalized column vector in $\mathbb{C}^n$. Using Dirac notation [57], we denote such a vector as $|\phi\rangle$. To do a computation, we must have ways of altering states. Most often, we transform a state using a unitary transformation, i.e., an action on a state $|\phi\rangle \in \mathbb{C}^n$ can be regarded as a unitary matrix $U : \mathbb{C}^n \mapsto \mathbb{C}^n$. After computation, the state becomes $U|\phi\rangle$. The sequential application of two transformations U followed by V yields the state $V(U|\phi\rangle)$, or $(VU)|\phi\rangle$. Once the system reaches a solution state $|\phi'\rangle$, we can extract the encoded information by applying a quantum measurement operator.

This chapter briefly introduces the basic concepts of quantum computing required to understand the contents of the thesis. For more details, we refer the reader to [105].

2.1. QUANTUM BITS

The quantum bit, or *qubit* for short, is the unit of quantum information. The state of a qubit is described as a unit vector in $\mathbb{C}^2$. As is customary, we fix an orthonormal basis $\{|0\rangle, |1\rangle\}$ of $\mathbb{C}^2$ called the computational basis. The states $|0\rangle$ and $|1\rangle$ are known as classical states. A qubit is fundamentally different from a bit because, until measured, it can be on a state of superposition of classical states—i.e., a linear combination of both classical states:

$$|\phi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

where α and β are complex values, known as amplitudes. Since the state of a qubit is required to be a unit vector in $\mathbb{C}^2$, we have $|\alpha|^2 + |\beta|^2 = 1$. Furthermore, note that each squared amplitude $|\alpha|^2$ and $|\beta|^2$ indicate the probability of whether the qubit state will be classical 0 or 1 after a measurement operation, respectively.

The state of n qubits is given by a unit vector in the 2^n-dimensional complex vector space $\mathbb{C}^{2^n}$. We denote the states of the computational basis by $|\boldsymbol{x}\rangle = |x_n \dots x_2 x_1\rangle$, where $\boldsymbol{x} \in \mathbb{F}_2^n$ and $\mathbb{F}_2 = (\{0,1\}, \oplus, \cdot)$ is the binary field. We can combine the state of different subsystems $|\phi_0\rangle \in \mathbb{C}^{2^n}$ and $|\phi_1\rangle \in \mathbb{C}^{2^m}$ by taking their tensor product $|\phi_1\rangle \otimes |\phi_0\rangle \in \mathbb{C}^{2^{m+n}}$, which is defined as:

$$|\phi_1\rangle \otimes |\phi_0\rangle = \sum_{\boldsymbol{y} \in \mathbb{F}_2^m} \beta_y |y\rangle \otimes \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} \alpha_x |x\rangle = \sum_{\boldsymbol{y} \in \mathbb{F}_2^m} \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} \alpha_{\boldsymbol{x}} \beta_{\boldsymbol{y}} (|\boldsymbol{y}\rangle \otimes |\boldsymbol{x}\rangle),$$

where $|\boldsymbol{x}\rangle$ and $|\boldsymbol{y}\rangle$ are orthonormal basis for the spaces $\mathbb{C}^{2^n}$ and $\mathbb{C}^{2^m}$, respectively. To simplify the notation, we often omit the $\otimes$ sign when computing the tensor product of two basis states $|\boldsymbol{x}\rangle$ and $|\boldsymbol{y}\rangle$. The resulting state is defined as the basis state labelled by the concatenation $\boldsymbol{yx}$:

$$|\boldsymbol{y}\rangle \otimes |\boldsymbol{x}\rangle = |\boldsymbol{yx}\rangle = |y_m \dots y_2 y_1 x_n \dots x_2 x_1\rangle .$$

Example 2.1. *Suppose we have two-qubit system in states $\alpha|0\rangle + \beta|1\rangle$ and $\gamma|0\rangle + \delta|1\rangle$, respectively. The combined state is given by:*

$$\begin{aligned}(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) = \alpha\gamma(|0\rangle \otimes |0\rangle) + \alpha\delta(|0\rangle \otimes |1\rangle) \\ + \beta\gamma(|1\rangle \otimes |0\rangle) + \beta\delta(|1\rangle \otimes |1\rangle)\end{aligned}$$

Using the simplified notation, We can rewrite it as:

$$\alpha\gamma|00\rangle + \alpha\delta|01\rangle \ + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

A multi-qubit state that can be written as a combination of one-qubit states is said to be separable. A non-separable state is said to be entangled. For example, no two independent one-qubit states $|\phi_0\rangle$ and $|\phi_1\rangle$ exists such that $|\phi_1\rangle \otimes |\phi_0\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$—a state known as the Bell state, a perfect superposition of the classical states 00 and 11.

2.2. QUANTUM OPERATORS

A quantum operator U maps one quantum state into another, i.e, $U : \mathbb{C}^{2^n} \mapsto \mathbb{C}^{2^n}$, where n is the number of qubits. A quantum unitary operator, or just unitary, is an invertible linear operator such that $U^\dagger = U^{-1}$, where $U^\dagger$ is the conjugate-transpose (adjoint) of U. Equivalently, U is a linear operator that preserves the L_2-norm, and thus maps unit vectors to unit vectors. We denote $\mathscr{U}(d)$ the set of unitary operators on a complex vector space of dimension d. After computation, the state becomes $U|\phi\rangle$. The sequential

Name(s)	Symbol(s)	Matrix
Identity	Id	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Hadamard	H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Pauli-X, Not	X, σ_x	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y	Y, σ_y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z	Z, σ_z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Phase	P(θ), $R_1(\theta)$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$
S-gate, $\sqrt{\text{Z}}$	S	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
T-gate, $\pi/8$	T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$
Controlled Not	CNOT	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
Swap	SWAP	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Table 2.1: Common quantum operators by name(s), symbol(s) and the corresponding unitary matrices.

application of two transformations $U, V \in \mathscr{U}(2^n)$ yields the state $V(U|\phi\rangle)$, or $(VU)|\phi\rangle$—where U is applied first. Two unitary operators $U \in \mathscr{U}(2^n)$ and $W \in \mathscr{U}(2^m)$ operating on different subsystems may be combined with the tensor product $U \otimes W \in U(2^{n+m})$:

$$(V \otimes U)(|\phi_1\rangle \otimes |\phi_0\rangle) = V|\phi_1\rangle \otimes U|\phi_0\rangle\,.$$

Unitary operators can fully describe quantum computations. However, they are of no practical use by themselves since we cannot directly access all the information describing quantum states—in particular, given the state of a qubit $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, we cannot measure the complex amplitudes α and β. This lack of access is not just a practical limitation; it is part of the postulates of quantum physics. Measurement is an operation that allows reaching into the Hilbert space to probe the quantum state. A crucial aspect

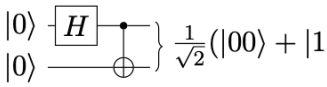
of it is that it alters the state of the quantum system: the effect of the measurement is that the new state is precisely its outcome; in the case of our qubit, either $|0\rangle$ or $|1\rangle$. Therefore, measurement is a non-unitary operator.

2.3. THE CIRCUIT MODEL

The quantum circuit model of computation introduced by Deutsch [54] and further developed by Yao [39], provide us with a convenient tool for representing the evolution of a quantum system. For example, if we want to describe the computation the Bell state for two-qubits system initialized in state $|00\rangle$, we may write

$$\mathrm{CNOT}((H \otimes I)\,|00\rangle) = \frac{1}{\sqrt{2}}\,|00\rangle + \frac{1}{\sqrt{2}}\,|11\rangle\,.$$

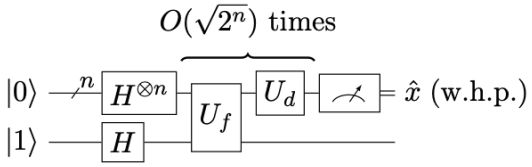
We first apply a Hadamard operation on the first qubit and then apply the CNOT operator to the resulting state and the second qubit. The shortcomings are obvious: the order of operations reads from right to left. We need to explicitly specify all identity operations to get the padding-right to perform matrix multiplication. The quantum circuit model allows us to write the same computation as



.

This diagram is read from left to right, with each horizontal line representing a qubit. The horizontal axis indicates time. A box $\boxed{U}$ on the qubits' lines expresses the application of an operator U. Some operators have a special notation. For example, the '$\oplus$' symbol represents the X operator. Also, a solid circle '$\bullet$' indicates the control qubit(s) of a controlled operator.

We can use the quantum circuit model to very concisely represent some quantum algorithms. As one example, the following quantum circuit describes the Grover algorithm [69]. For an n-variable Boolean function f for which there exists exactly one $\hat{x}$ such that $f(\hat{x}) = 1$, it can find $\hat{x}$ with high probability by querying f only $O(\sqrt{2^n})$ times.



The above circuit first creates a uniform superposition of all inputs. Then it iterates over a quantum implementation of f (i.e., $\boxed{U_f}$), that marks the solution state $|\hat{x}\rangle$, and a quantum operator $\boxed{U_d}$, known as diffusion, amplifies this state's amplitude. Finally, since we amplified the amplitude of the solution state, the final measurement will return the solution with high probability.

2.4. REPRESENTATION OF QUANTUM FUNCTIONALITY

The basic mathematical objects to be dealt with when representing quantum functionality are Hamiltonians of a quantum system. These are linear, unitary mappings $\mathbb{C}^{2^n} \mapsto \mathbb{C}^{2^n}$ that describe the system's evolution. There are several ways for representing quantum operators, and each way has its strengths and weaknesses. We evaluate the efficiency of a representation by its succinctness in describing operators and its capability of supporting transformations and manipulations. Since no representation is universally suitable for all applications, conversion is essential during compilation, where various synthesis and transformation techniques are applied.

Unitary matrix. The most natural way to represent a Hamiltonian is to choose a basis of the Hilbert space and then consider the corresponding transformation matrix, which is a $2^n \times 2^n$ complex-valued unitary matrix. Unitary matrices are canonical representations of quantum operators. That is, two quantum operators are equivalent if and only if they have the same unitary matrix. Canonicity is an important property that may be useful in many applications of synthesis and verification. Unitary matrices, however, are often impractical to represent operators that act on many qubits because their size grows exponentially with the number of qubits. For instance, if we use a unitary matrix to represent a functionality on IBM's 65-qubit computer, it would have $2^{65} \times 2^{65} \approx 1.361129 \cdot 10^{39}$ entries. (Also, if we had the means to represent big unitary matrices explicitly, then we would not need quantum computation.) Though theory allows arbitrary unitary matrices, the currently available physical hardware can only handle a limited set operating on one or two qubits.

Example 2.2. *The Toffoli gate, named after Tommaso Toffoli [149], is a 3-qubit gate, which is almost*[1] *universal for classical computation but not for quantum computation. Its matrix representation is:*

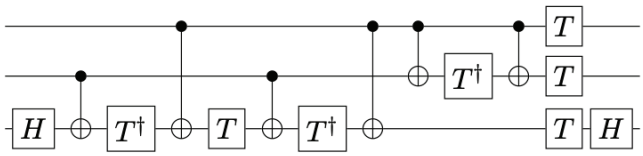
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Circuit. As discussed in the previous subsection, quantum circuits provide a convenient tool for representing quantum computations. While such representation has the

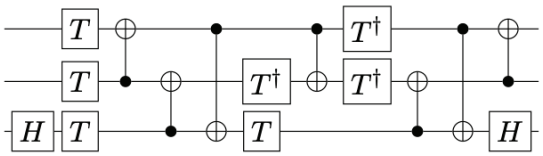
[1] Various sources wrongly classify the Toffoli operator as universal (or functional complete). However, this cannot be the case since it belongs to the set of falsity-preserving operators [109].

advantage of simplicity, it lacks canonicity, i.e., there are many different ways of representing a given computation with an available set of universal elementary operators. Finding an implementation that uses the fewest resources is not only advantageous but imperative given the stringent resource constraints in quantum hardware.

Example 2.3. *The following circuit implements the Toffoli operator using the Clifford+T gate set:*



As we mentioned, circuits are not a canonical representation. Thus there are various other ways of implementing the Toffoli operator. For example, the following construction can be found in [16]:



While this thesis focuses on quantum circuits, one should know that many other ways of representing quantum functionality exist, e.g., quantum decision diagrams [99, 5, 106], phase polynomials [19], and ZX calculus [42, 43], to name a few. If we consider restricted functionality, the list gets even longer. This is because no representation is universally suitable for all applications, and thus, as we demonstrate in Chapter 9, conversion between them is crucial for developing robust compilation flows for quantum programs.

2.5. ORACLES

A *quantum oracle* is a "black-box" operator that is used as an input to another algorithm. Such an oracle can often be understood as a classical computation specified by a Boolean function. Oracles are widely used for studying the complexity of quantum algorithms [15]. Counting the number of oracle queries needed to evaluate a function is easier than counting the number of computational steps. Thus, to try inferring nontrivial lower bounds more readily, quantum algorithm researchers characterize the computational complexity of an algorithm by the asymptotic growth rate of the number of queries with growing input size—an analysis that only assumes the existence of an oracle, and does not require its implementation.

We say that the oracle gives access to a Boolean function, meaning that an algorithm that uses the oracle only has access to the function's input and output, not its internal structure. In the following, we describe two natural ways of implementing an oracle characterized by a Boolean function f on a quantum computer.

Bit oracle. A bit oracle is a quantum operator B_f specified by a Boolean function f for which the effect on all computational basis states is given by

$$B_f : |x\rangle |y\rangle |0\rangle^a \mapsto |x\rangle |y \oplus f(x)\rangle |0\rangle^a , \tag{2.1}$$

where '$\oplus$' is the logical exclusive-or operator and $a \geq 0$ corresponds to the number of extra qubits used to store intermediate results for the computation of $f(x)$, the so-called *ancilla qubits*.

Phase oracle. A phase oracle is also a quantum operator specified by a Boolean function f. However, its effect on all computational basis states is given by

$$P_f |x\rangle = (-1)^{f(x)} |x\rangle . \tag{2.2}$$

Equation (2.2) means that if x is not a satisfying input, the oracle does nothing to its corresponding state $|x\rangle$. Otherwise, it rotates the states' phase by π (or 180 degrees).

It turns out that these two oracle models are almost equivalent: the phase oracle can be obtained from one use of the bit oracle and the use of the Hadamard operator on the output qubit. There's also a subtlety that $f(x)$ and $\bar{f}(x)$ are equivalent for phase oracles and cannot be distinguished because of a global phase.

Part II

Synthesis

Chapter 3

Introduction to quantum circuits synthesis

A look into the "Quantum Algorithm Zoo" website [76] reveals that most quantum algorithms are specified in a high level of abstraction. These specifications use quantum operators defined as unitary matrices or other abstract representations non-executable by the hardware directly. Hence, one has to find a way to turn these operators into a quantum circuit made of elementary gates. This problem is known as *circuit synthesis* and can be either exact or approximate with a certain precision ϵ —i.e., given an operator U, we can synthesize any circuit C such that $\|U_C - U\| \leq \epsilon$, where U_c is the unitary that circuit implements.

Recall that circuits are not canonical, i.e., many quantum circuits can implement a unitary over a given set of elementary operators. Hence, we can choose, or search for, the one that minimizes a precise criterion. There are three widely used criteria to evaluate synthesis algorithms' quality of results:

- The *size* of the resulting circuit in the number of elementary gates. The rationale behind this criterion is that one cannot avoid some noise in the physical realization of the gate, and noise can alter the output of a computation to a wrong result.

- The *depth* of the resulting circuit in the number of time steps. The depth is closely related to the time required for the execution of the quantum circuit. One significant problem physicists face when designing a quantum computer is the decoherence time, i.e., maximum time available to execute a quantum algorithm before the qubits involved in the computation interact enough with the outside environment to lose all the information they carry.

- The *width* of the resulting circuit in the number of qubits. Some synthesis algorithms require the use of extra quantum memory, i.e., the use of ancillary qubits. Given the stringent qubit constraints in current quantum hardware, finding circuits that use fewer qubits is not only advantageous but imperative. Also, ancillary qubits often help reduce the depth and the number of costly operations when using

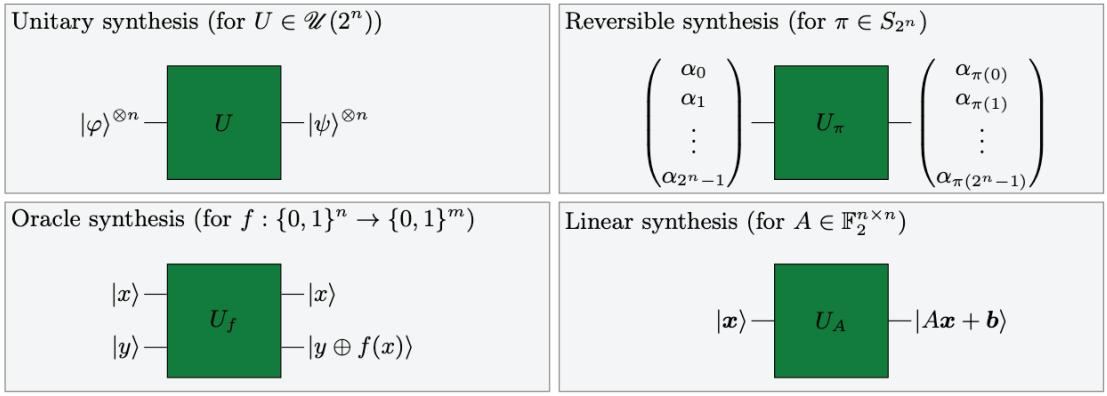


Figure 3.1: Some common quantum synthesis problems that emerge during compilation of quantum programs.

a fault-tolerant gate set, e.g., reducing the number of T gates in circuits over the Clifford+T gate set.

3.1. SYNTHESIS PROBLEMS

Synthesis problems and their respective solving techniques change drastically depending on the abstract representation of the quantum functionality we want to implement. For example, the synthesis of generic operators is exponentially hard by nature: being generic means that we are dealing with unitary matrices without any particular structure, which, in turn, implies that having complete knowledge of such an operator requires exponentially sized memory and an equivalently exponential amount of time access each all entries. Therefore, even when heuristic, techniques that solve this problem work only for operators with few qubits.

Reversible Synthesis. If an operator consists of a permutation of the basis states, its matrix representation is a permutation matrix of size $2^n \times 2^n$, and we can describe its behavior using a reversible Boolean function $f_r : \{0,1\}^n \mapsto \{0,1\}^n$, where f_r is a bijective function. A quantum circuit that implements this function realizes the unitary

$$U_\pi : |x\rangle \mapsto |f_r(x)\rangle \,.$$

The function $f_r(x)$ can come in many forms, such as a truth table, a permutation, a decision diagram, or a logic network. The realm of reversible logic synthesis deals with the problem of synthesizing a reversible circuit out of one of these forms. Most early algorithms expected a truth table [97, 90, 52, 115] or a permutation [127] as input. Since these representations grow exponentially with n, these algorithms do not scale well. Thus, they not efficiently applicable to large reversible functions, i.e., $n > 20$. To cope with this problem, researchers proposed alternative implementations based on symbolic representations, e.g., binary decision diagrams [140, 139] or Boolean satisfiability prob-

lems [133]. While these symbolic representations can sometimes handle large functions, they do not always guarantee compactly representing them.

Oracle synthesis. This is a particular case of reversible synthesis where we want to implement a specific irreversible Boolean function $f : \{0,1\}^n \mapsto \{0,1\}^m$. Note that real-world problems and algorithms of interest almost exclusively use irreversible functions. Since a quantum circuit cannot represent such functions, we must embed f into a reversible function—a process doable either implicitly or explicitly. For the latter, we need to find a reversible function f_r over k variables such that

$$f_r(x, y) = (x, y \oplus f(x)),$$

where $x = x_0, \ldots, x_{n-1}$, $y = y_n, \ldots, y_{k-1}$, $k \geq \max(n, m)$, and '$\oplus$' referring to the XOR operation. Such an embedding is also referred to as Bennett embedding [24], and implies the existence of the following quantum operation:

$$U_f : |x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle .$$

The function f can be given in various forms, e.g., a logic network or a decision diagram. If the function is small, we can directly synthesize a quantum circuit by representing it as an ESOP. The following chapter demonstrates how we can push the boundary of what is considered "small."

Nevertheless, we need to frequently employ a hierarchical synthesis approach to deal with practical functions. Such an approach might even rely on a direct method to synthesize a circuit. They work by decomposing a function and storing intermediate results on ancilla qubits. Given an irreversible Boolean function f they find an $(n+1+a)$-qubit quantum circuit that realizes the unitary

$$U_f : |x\rangle |y\rangle |0\rangle^a \mapsto |x\rangle |y \oplus f(x)\rangle |0\rangle^a$$

where $a > 0$, which means that the synthesis algorithm might use the a additional qubits to store intermediate computations. Chapter 5 presents improvements to a state-of-the-art technique that directly synthesizes a circuit from a XAG. While scalable, a significant disadvantage of such methods is that the number of ancilla qubits cannot be bounded apriori, i.e., it is a consequence of the algorithm's execution. Therefore, a hard limit on the number of qubits might restrict their use. There are few techniques to mitigate this problem [96, 56], enabling the exploration of the trade-off qubits and gates.

Linear Synthesis. Linear synthesis is another particular case of reversible synthesis. In this case, we are interested in operators that belong to the family of linear reversible circuits, i.e., CNOT circuits. Over the years, researchers have extensively studied their irreversible counterparts [13, 107, 64, 31]. Abstractly, we can represent a linear operator acting on n qubits as a Boolean matrix $A \in \mathbb{F}_2^{n \times n}$ where each row corresponds to one

output as a linear combination of the inputs, and $\mathbb{F}_2$ is the Galois field of two elements. In our case, A needs to be invertible, and CNOT is the primitive linear gate. Unfortunately, such constraints make difficult the reuse of known synthesis techniques from the irreversible realm in the reversible (quantum) one.

In [108], Patel et al. designed a block version of the Gaussian elimination algorithm to synthesize linear reversible circuits. More recently, two papers proposed modifying the Gaussian elimination algorithm to synthesize circuits compliant with a connectivity graph given as input of the algorithm [79, 104]. Both use Steiner trees to perform a custom Gaussian elimination: the circuit is synthesized column by column, but the process to eliminate nonzero elements is modified to respect the connectivity constraints.

In the context of quantum circuit mapping, we are interested in synthesizing a special family of linear circuits: the permutation circuits, which must also respect coupling constraints. These circuits appear prominently in various quantum computing benchmarks [49] and might dominate the their total implementation costs.

Chapter 4

ESOP-based synthesis

4.1. MOTIVATION

This chapter investigates ways of scaling-up ESOP-based synthesis for oracle synthesis. The importance of this investigation resides in the severe resource limits imposed by near-term quantum hardware. While there are various techniques to synthesize a quantum circuit from a Boolean network, most require an unbounded number of additional qubits, i.e., the algorithm's execution determines the total number of qubits in the resulting circuit.

In contrast, due to a natural affinity between exclusive sum-of-products (ESOP) expressions and Toffoli gates, ESOP-based synthesis can generate reversible circuits with an optimal number of qubits. Meaning that given an ESOP expression for a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}^m$, one can readily derive a reversible network that acts on $n + m$ qubits.

A major drawback in using this technique lies in its required input: a two-level ESOP expression. In practical applications, we rarely find Boolean functions represented as ESOP, especially the large ones. They are often given as multi-level Boolean networks, and thus we need to convert them into ESOP expressions through a process known as *collapsing*. There are two state-of-the-art techniques to collapse a Boolean network represented as an and-inverter graph (AIG) into a two-level ESOP expression: the AIG extract method and the BDD extract method [137]. In this chapter we study them in detail.

4.2. CONTRIBUTIONS

Both state-of-the-art techniques lack the necessary scalability to cope with the increasing complexity of the logic functions used on quantum computers. Therefore, in this chapter, we present the results of research directed towards the development of a new ESOP extraction algorithm:

- We revisit both the AIG and BDD extract methods and propose ways to improve them.

- We introduce a divide-and-conquer collapsing method, called *DC extract*, that overcomes scalability limitations of the state-of-the-art approaches.

- We apply the new collapsing method in an ESOP-based reversible logic synthesis technique, allowing us to find quantum circuits with up to 50% reduced quantum gate costs.

The experimental results at the end of this chapter confirm the effectiveness of the proposed method. Specifically, we show that we are able to collapse AIGs that previous methods were unable to. This opens up the possibility of a qubit-optimal realization of these circuits, which is crucial due to the severe resource limits imposed by today's and near-term quantum hardware.

This work appears in [123] and was presented at IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL'2019). An implementation appears in `tweedledum`.

4.3. PREVIOUS WORK

Before describing our approach to collapsing, we present two methods previously used, namely AIG [137] and BDD [58] extract. We explain some of their implementation details, respective shortcomings and examine ideas on improving those techniques. We show that these ideas did not lead to sufficient improvements and thus further affirm the inability of the two approaches to cope with the complexity of large benchmarks.

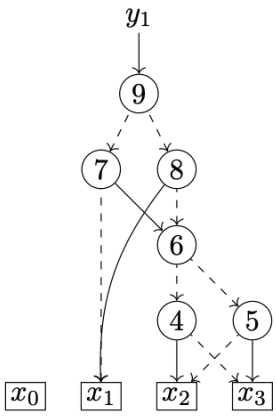
AIG extract. This method computes an ESOP expression for each vertex in an AIG in a topological order. First, each primary input x_i is assigned the trivial ESOP expression x_i. The ESOP expression of each subsequent vertex is computed by conjoining the ESOP expressions of its previous nodes:

$$e_i = \begin{cases} x_i & \text{if } 1 \leq i \leq n, \\ (e_{i_0} \oplus p_{i_0}) \wedge (e_{i_1} \oplus p_{i_1}) & \text{if } n+1 \leq i \leq n+r, \end{cases} \tag{4.1}$$

where r is the number of gates, i_0 and i_1 identify x_i inputs, $1 \leq i_0 < i$ and $1 \leq i_1 < i$. The polarity flags p_{i_j} are Boolean constants used to complement the gate's fan-ins and thus used to complement the ESOP expression corresponding to these fan-ins. The following example illustrates the technique.

Example 4.1. *The following AIG implements $f = x_1 \oplus x_2 \oplus x_3$. Recall from Chapter 1 that in a AIG, each vertex is a 2-input AND gate. In this example, we chose to represent them with their index i inside instead of the operator $\wedge$ to better relate them to their corresponding ESOP expression e_i.*

$$e_1 = x_1 \qquad e_2 = x_2 \qquad e_3 = x_3$$
$$e_4 = x_2\bar{x}_3$$
$$e_5 = \bar{x}_2 x_3$$
$$e_6 = x_2\bar{x}_3 \oplus \bar{x}_2 x_3 \oplus 1$$
$$e_7 = \bar{x}_1 x_2\bar{x}_3 \oplus \bar{x}_1\bar{x}_2 x_3 \oplus \bar{x}_1$$
$$e_8 = x_1 x_2\bar{x}_3 \oplus x_1\bar{x}_2 x_3$$
$$e_9 = \bar{x}_1 x_2\bar{x}_3 \oplus \bar{x}_1\bar{x}_2 x_3 \oplus x_1 x_2\bar{x}_3 \oplus x_1\bar{x}_2 x_3 \oplus x_1$$



Above, we show the individual ESOP expression for each vertex of the AIG. We compute these expressions using (4.1)*. Since expressions* e_1 *through* e_5 *are straightforward to obtain, we workout show how to calculate expression* e_6*:*

$$\begin{aligned} e_6 &= (e_4 \oplus 1) \wedge (e_5 \oplus 1) \\ &= e_4 e_5 \oplus e_4 \oplus e_5 \oplus 1 \\ &= \cancelto{0}{x_2\bar{x}_3\bar{x}_2 x_3} \oplus x_2\bar{x}_3 \oplus \bar{x}_2 x_3 \oplus 1 \\ &= x_2\bar{x}_3 \oplus \bar{x}_2 x_3 \oplus 1 \end{aligned}$$

As shown in the above example, the number of product terms in the resulting ESOP expression of each vertex is the product of the number of terms of its inputs. Hence, the usage of this technique is highly limited in both scalability and quality of results. Note the difference in the size of the computed ESOP expression and the original function.

We can slightly improve this collapsing technique by optimizing the ESOP expressions generated through its application. Over the years, researchers have proposed several ESOP minimization strategies [103, 113]. However, none is scalable enough to be applied after collapsing each vertex without paying a significant execution time penalty. Thus, we use a greedy, low-effort minimization strategy in our implementation. The strategy relies on properties of ESOP expression:

- We can add two identical terms, i.e., distance-0, to any ESOP without changing the function represented by it,
- A single term can represent the XOR of two distance-1 cubes.

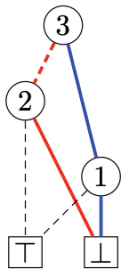
One can think of our technique as a "lazy" exorcism [103]: before adding a new term to an expression, it first tries to find another term that is either distance-0 or distance-1. If this search is successful, then it transforms both terms accordingly. Despite its simplicity, empirical observations indicate that most of the time, the size is about the same as the size of its inputs largest ESOP.

Parallelization is another way of improving this method. Indeed, this technique is embarrassingly parallel since we traverse the logic network in topological order. The

construction of each internal vertex's ESOP depends only on ESOPs of vertices that we have already visited and thus already have an ESOP for it. Hence, we can collapse all vertices in the same level in parallel. Our work, however, does not explore such improvement.

BDD extract. This method first expresses the AIG as a BDD by translating each vertex into a BDD in topological order. From the BDD, a particular case of an ESOP expression, a Pseudo-Kronecker expression (PSDKRO) [117], is extracted using the algorithm presented in [103]. For a detailed discussion of the algorithm, we refer to [58].

Before delving into the details of how to extract a Pseudo-Kronecker expression from a BDD, let me show how we can solve a more straightforward problem, namely, how to generate an SOP, or a general ESOP, from a BDD. Recall from Subsection 1.2.3 that using a binary decision diagram, one can determine the value of a Boolean function for any given variable assignment by following a path from the root vertex to a terminal vertex. We can use this fact to generate an SOP expression by focusing on paths that terminate at the TRUE vertex. Each of these paths corresponds to one SOP term. We construct the terms by following the paths from the root to TRUE while collecting the factors[1].



$$f = \bar{x}_3 x_2 \vee x_3 x_1 \tag{4.2}$$

Also, note that in $f = x_i f_{x_i} \vee \bar{x}_i f_{\bar{x}_i}$, the two terms are mutually disjoint, and thus we can trivially replace '$\vee$' with '$\oplus$', yielding the XOR-form of Boole's expansion. Since we use this expansion to build a BDD, an SOP generated by traversing the BDD has the interesting property that all its cubes are also mutually disjoint, i.e., they cover disjoint sets of minterms. Therefore, we can trivially transform the SOP in (4.2) into an ESOP by replacing $\vee$' with '$\oplus$', thus

$$f = \bar{x}_3 x_2 \vee x_3 x_1 = \bar{x}_3 x_2 \oplus x_3 x_1. \tag{4.3}$$

Technically, the ESOP in (4.3) is a Pseudo-Kronecker expression; however, not a good one. Since we built it by traversing the BDD, we implicitly considered only the XOR-form of Boole's expansion. We can achieve better results by also considering two

[1]Keep in mind that an algorithm that extracts an SOP from a BDD does not know a priori the paths that lead to the terminal vertex TRUE. Hence, it follows all paths while collecting factors. If the path leads to TRUE, a cube is created using these factors. Otherwise, they are discarded

other expansions:

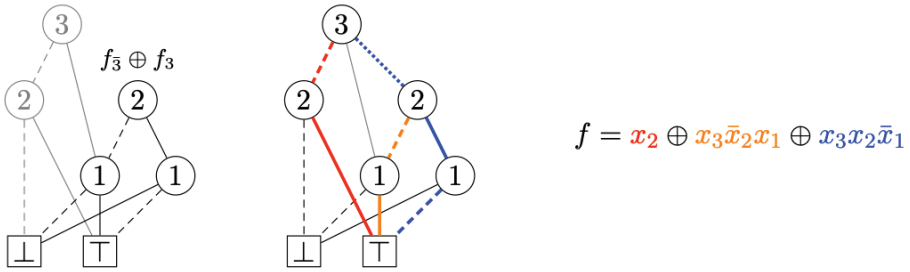
$$f = f_{\bar{x}_i} \oplus x_i \frac{\partial f}{\partial x} \qquad (positive\ Davio) \tag{4.4}$$

$$f = f_{x_i} \oplus \bar{x}_i \frac{\partial f}{\partial x} \qquad (negative\ Davio) \tag{4.5}$$

where $\frac{\partial f}{\partial x}$ is the Boolean derivative, or Boolean difference, of f with respect to x, defined as $f = f_{\bar{x}_i} \oplus f_{x_i}$.

In some sense, when we follow both paths, $f_{\bar{x}_i}$ and f_{x_i}, of a decision vertex i while collecting the factors, $\bar{x}_i$ and x_i, we are applying Boole's expansion. Now, if we want to use the positive Davio expansion instead, we must follow the $f_{\bar{x}_i}$ path without collecting the factor and simulate following a path to $\frac{\partial f}{\partial x}$ while collecting x_i. Note, however, that $\frac{\partial f}{\partial x}$ might not be present in our decision diagram, and thus we must add it. We use this same principle to use the negative Davio expansion.

Example 4.2. *Suppose we are given a Boolean function f as the BDD in (4.2), and we want to extract an ESOP expression from it. Our idea is to use the positive Davio expansion once and Boole's expansion twice. We must first add $\frac{\partial f}{\partial x_3}$ to the diagram to successfully use our idea. Then we traverse the BDD following the path corresponding to the expansions we chose.*



Note that is this case, we did not follow all paths in the BDD, and the dotted path from (3) to (2) does not actually exist—its a "simulated" path.

The ESOP expression obtained in the previous examples is larger than in (4.3). Clearly, our chosen sequence of expansions was not particularly good. We can try changing the sequence to find a better ESOP, which the BDD extract algorithm cleverly does. The algorithm traverses the BDD twice. The best expansion is found for each node during the first pass and saved in a hash table. During the second pass, we follow the paths corresponding to the best expansion of each vertex while collecting the factors. When the traversing procedure reaches the bottom of the diagram in the TRUE vertex, it uses all collected factors to generate a cube and add it to the resulting ESOP expression.

Implementing BDD extract is reasonably straightforward using a BDD package such as CUDD [141]. We enable dynamic variables reordering during the BDD construction, which allows us to work with relatively large multi-output functions. Nevertheless, it is the construction process that bounds the performance. Furthermore, there exists a family of Boolean functions whose BDD sizes are exponential in their formula sizes under all

variable orderings. This unfortunate characteristic of BDD causes this extraction method not to have the necessary scalability to cope with the increasing complexity of the logic functions used on quantum computers.

4.4. DIVIDE-AND-CONQUER EXTRACTION

The main technical contribution of this chapter is the introduction of a divide-and-conquer collapsing method. This new approach stands on a natural and straightforward idea:

- Divide the given problem into small-enough subproblems.
- Solve these subproblems independently.
- Combine these solutions to get a final solution.

Our divide-and-conquer technique relies on the fact that any Boolean function f can be written as

$$f = f \wedge g_1 \oplus f \wedge g_2 \oplus \cdots \oplus f \wedge g_n \tag{4.6}$$

for n Boolean functions g_n over the same support as f. By factoring, the right-hand-side becomes

$$f \wedge (g_1 \oplus \cdots \oplus g_n)$$

which equals f when $g_1 \oplus \cdots \oplus g_n = 1$. The careful selection of a set $\{g_1, g_2, \ldots, g_n\}$ is a key component of our method because it directly affects its effectiveness. We need to choose this set in such a way that makes easier to collapse *all* individual $f \wedge g_i$. This might sound counter-intuitive at first, but if we limit our choices for g_i so that we have a set of single-cube factors of f, then the variables present in g_i can be ignored when collapsing $f \wedge g_i$, i.e., we will collapse f_{g_i} instead of the whole f.

In practice, our implementation represents f as an AIG and g_i as cube; $f \wedge g_i$ is represented as a tuple (f_{g_i}, g_i), where f_{g_i} is the original AIG f with the variables present in g_i being assigned constant values in accordance to their polarity in the cube. These constant values simplify the AIG through constant propagation.

After dividing the problem, we use BDD extract to collapse these f_{g_i} AIGs into ESOP e_i. Finally, we can combine our results in two different ways: one will leave the factors multiplying the intermediate ESOPs, while the other will expand the multiplications. The following example illustrates our method.

Example 4.3. *Let f be a 3-variable Boolean function. Using* (4.6)*, we can represent the function as*

$$f = f \wedge g_1 \oplus f \wedge g_2 \oplus f \wedge g_3 \oplus f \wedge g_4$$

with $g_1 = x_1 \wedge x_2$, $g_2 = \bar{x}_1 \wedge x_2$, $g_3 = x_1 \wedge \bar{x}_2$ and $g_4 = \bar{x}_1 \wedge \bar{x}_2$.

With f given as an AIG, we can divide our problem by creating the four tuples (f_1, g_1), (f_2, g_2), (f_3, g_3), (f_4, g_4). We solve it by collapsing the simplified f_i separately, where

$i = \{1, 2, 3, 4\}$, thus generating: (e_1, g_1), (e_2, g_2), (e_3, g_3), (e_4, g_4). Finally, we can report the combined solution either as:

$$f = e_1 \wedge g_1 \oplus e_2 \wedge g_2 \oplus e_3 \wedge g_3 \oplus e_4 \wedge g_4$$

or as an expanded ESOP expression, with the multiplications $e_i \wedge g_i$ carried out.

4.4.1 Selection heuristics

The efficiency of our method is intimately related to the selection of variables to factor. Therefore, this section defines different heuristics to select the variables.

Fixed variable selection. This heuristic chooses variables to factor using only one criterion: the combined size, in the number of vertices, of the resulting AIGs. The idea is to minimize this number as much as possible. The first variable is chosen by temporarily factoring all variables, one at a time, and choosing one with minimal combined size. Using the two resulting AIGs as starting point, we select the second variable in the same way: we temporarily factor all remaining variables, one at a time, from both AIGs and pick a variable using the same criteria. This process continues until we have chosen a predefined number of variables.

Free variable selection. This heuristic selects variables using the same criteria as the previous one. The difference is the recursive way it chooses them. After choosing the first variable, the two resulting AIGs are used as two different starting points to choose the second one. In practice, this means that the selected variable as the second factor in one branch might differ from that picked in the other.

4.5. EXPERIMENTAL RESULTS

We implemented the algorithm described in C++ using both CUDD and ABC [33] as external static libraries. ABC is an open-source tool designed for logic synthesis, technology mapping, and formal verification for logic circuits. We also use it to check the results for equivalence.

We use a set of Verilog netlists of several IEEE-compliant arithmetic floating-point designs in half (16-bit) precision to evaluate our method. For synthesis, all Verilog files were translated into AIGs and optimized for size using ABC's `resyn2` script. The collapsing of these functions is used as an example to illustrate the strength of this method when searching for a good ESOP representation which is a good starting point for a quantum logic synthesis flow.

We ran all experiments on an Intel(R) Xeon(R) CPU E5-2690 v4 at 2.60GHz and used the GNU 1.7 version of the command `time` to obtain the reported execution times and peak memory.

The results of collapsing are reported in Table 4.2. The first column identifies each benchmark by its name. The remaining columns report the results of using different techniques for collapsing in terms of ESOP number of cubes (# terms), time, and peak memory usage (mem).

Both state-of-the-art methods failed to collapse the whole set of benchmarks. In this experiment, we set the timeout to one week. Note that the AIG extract has the worst execution time and result quality. BDD extract, run with dynamic variable reordering, improves these results and can collapse an additional benchmark. The free variable selection heuristic configured DC extract to factor cubes with 2, 4, and 8 variables. The improvement is remarkable; when cubes of 8 variables are factored, not only were all benchmarks successfully collapsed but also *fp_and*, *fp_div* and *fp_sub* were processed within four minutes and using much less memory—when compared with the other runs of DC extract. Note *fp_mult* runs out of memory (OOM) in the other runs of DC extract.

Finally, to evaluate the impact of factoring without considering the variable selection heuristics, we implemented a random variable selection procedure that behaves similarly to the free variable selection heuristic. Still, instead of minimizing the size of the resulting AIGs at each recursion step, it randomly selects one of the remaining variables. We used this heuristic to collapse each benchmark a hundred times. Not surprisingly, the results shown in Table 4.1 indicate the need to select the variables carefully.

	DC extract (8 random factored vars)			
benchmark	min	max	avg	stdev
fp_add	98.88	4515.3	976.63	1355.33
fp_cmp	0.55	0.94	0.71	0.16
fp_div	202.08	425.5	265.17	80.97
fp_exp	5.58	7.4	6.27	0.50
fp_invsqrt	1.18	1.79	1.51	0.25
fp_ln	0.65	1.11	1.00	0.18
fp_log2	0.55	1.02	0.91	0.19
fp_mult	—	—	—	—
fp_recip	0.78	1.3	1.17	0.18
fp_sincos	0.42	0.95	0.77	0.17
fp_sqrt	0.32	0.81	0.70	0.20
fp_square	0.3	0.66	0.50	0.15
fp_sub	150.61	2586.4	822.07	715.83

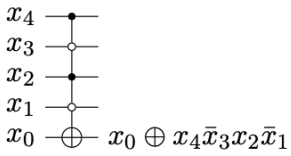
Table 4.1: DC extract execution time when choosing factors randomly.

	AIG extract			BDD extract			DC extract (2 factored vars)			DC extract (4 factored vars)			DC extract (8 factored vars)		
benchmark	# terms	time (s)	mem (Mb)	# terms	time (s)	mem (Mb)	# terms	time (s)	mem (Mb)	# terms	time (s)	mem (Mb)	# terms	time (s)	mem (Mb)
fp_add		TO			TO		16762634	27289.42	80190.32	17352120	10423.72	17077.70	14600025	134.81	1789.19
fp_cmp		TO		184363	0.80	16.86	184347	0.58	16.48	184353	0.54	17.22	184687	0.71	16.57
fp_div		TO			TO		29504220	4246.47	6397.96	30650976	1594.69	4618.52	30110188	242.20	2478.21
fp_exp	291726	14697.99	825.28	11470	282.24	156.88	11214	90.81	74.28	11087	21.59	47.43	11266	5.96	21.72
fp_invsqrt	15907	74.05	35.12	2213	2.00	18.70	2238	1.50	15.54	2925	1.38	13.55	5188	0.93	18.80
fp_ln	49240	6.40	10.21	16724	1.87	74.55	16301	1.14	23.48	16600	1.09	14.26	24684	0.65	14.59
fp_log2	30476	3.04	7.61	10816	0.60	17.84	10822	0.41	14.79	11502	0.28	13.04	18833	0.45	14.57
fp_mult		TO			TO				OOM			OOM	94869392	62811.34	120460.29
fp_recip	17478	3.89	8.11	3483	0.39	13.20	3576	0.23	12.79	3619	0.21	12.82	7346	0.79	18.47
fp_sincos	35992	10.03	11.08	5901	0.79	18.97	6285	0.57	18.95	6127	0.28	15.55	8949	0.39	13.51
fp_sqrt	19150	8.91	7.26	1963	0.16	12.01	2058	0.14	13.07	2218	0.10	12.47	3968	0.25	13.61
fp_square	7256	0.23	2.66	2683	0.02	12.01	2731	0.02	12.44	2853	0.06	12.37	6834	0.23	13.25
fp_sub		TO			TO		16801296	30392.64	115557.24	16988606	7211.76	16467.87	14502087	134.65	1784.04

Table 4.2: Execution times for the half precision (16-bit) arithmetic floating point designs.

4.6. USE CASE: ESOP-BASED REVERSIBLE LOGIC SYNTHESIS

ESOP-based logic synthesis is heavily used in reversible logic synthesis due to the natural correspondence of product terms in ESOP expressions and Toffoli gates. Let X be the set of variables in the reversible circuit. Recall that a (multiple-controlled mixed-polarity) Toffoli gate has a (possibly empty) set of control lines, which are literals over X and one target line, which is a variable in X that is not a control line. The Toffoli gate will invert the variable assigned to the target line if, and only if, the polarity of all inputs to the control lines match the polarities specified for the gate.

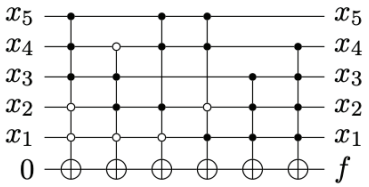


One can build up an ESOP expression on the same target line by concatenating Toffoli gates that act on the same target line. Vice versa, given an ESOP expression, one can easily extract a sequence of Toffoli gates with control lines according to the literals in the product terms. When considering multiple-output functions, the resulting quantum circuit requires $n + m$ qubits, where m is the number of outputs.

Example 4.4. *Given a Boolean function f such as*

$$f = \bar{x}_1\bar{x}_2x_3x_4x_5 \oplus \bar{x}_1x_2x_3\bar{x}_4 \oplus \bar{x}_1x_2x_4x_5 \oplus x_1\bar{x}_2x_4x_5 \oplus x_1x_2x_3 \oplus x_1x_2x_3x_4$$

Then the following reversible circuit implements f

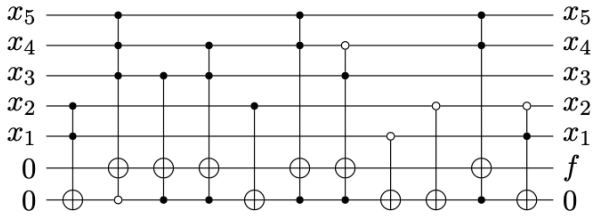


In quantum compilation, such a reversible circuit is an intermediate representation. Once we have it, we can straightforwardly map it into a lower-level quantum circuit over a given quantum gate library using additional steps. For example, we could map it to a universal fault-tolerant quantum gate library such as Clifford+T.

The cost model for quantum operations is non-trivial, but a good heuristic will try to minimize both the number of Toffoli gates and the number of controls in Toffoli gates, which corresponds, respectively, to the number of product terms in the ESOP expressions and the number of literals in the product terms. Such a heuristic is good because, when using the Clifford+T, the T gate is considered the most expensive, and the cost of implementing a Toffoli in the number of T gates is directly proportional to its number of controls. According to [92],the costs of implementing multiple-controlled Toffoli gates are

# Controls	0	1	2	3	4	5
# T gates	0	0	7	16	24	31

The circuit in example 4.4 has a cost of 143 T gates. We can do better using the proposed algorithm. In an ESOP expression generated through its use, we can cluster product terms with the same factor and use an additional helper qubit, called ancilla, to store these factor values. The following circuit illustrates this.



The bottom qubit is the ancilla, which we initialize to 0. We iteratively compute in this qubit all factors x_1x_2, $\bar{x}_1x_2$, $\bar{x}_1\bar{x}_2$, and $x_1\bar{x}_2$ in Gray code order—which allows us to transition from one factor to another using only CNOT gates. In general, if the factors have k literals, we can transition from one to another using a single $(k-1)$-controlled Toffoli gate, since the distance of two factors in Gray code order is 1. DC extract must use the fixed variable selection heuristic to exploit this property. The above circuit implements the same function as the one in example 4.4; while, at the expense of using one more qubit, it costs 109 T-gates.

4.6.1 Experimental results

We implemented the synthesis method described in previous section as a standalone tool capable of using the factored results of DC extract to synthesize reversible circuits. Table 4.3 shows the results in terms of the number of qubits and T gates. In the case of our technique, we also report the number of factored variables (v). The first column name the benchmarks. The next column presents the results obtained by directly mapping product terms into multiple-controlled Toffoli gates when the benchmark is collapsed without factoring, followed by the results of optimizing the direct mapping using [98], an approach which factors Toffoli gates that share the same controls and stores the factor on an ancilla qubit. The third column corresponds to results obtained when using the our synthesis method. As these experimental results show, our method of deriving the factors more explicitly, when compared to [98], leads to circuits with fewer quantum gates.

Finally, in the last column are the results of [137], which is a complete synthesis flow for quantum—hence our methods should not be viewed as a substitute but as a complement for it. Nevertheless, for the unitary operator benchmarks, we can achieve an improvement of up to 50% when compared to [137]. However, the results for the binary operators, namely *fp_add*, *fp_sub*, *fp_div*, and *fp_cmp*, have very large T gate count. The problem lies with the large size of their ESOP expressions, which, to our knowledge, cannot be handled by any state-of-the-art ESOP optimization tool. Consequently, we did

not use the direct method nor [98] to synthesize them. However, we expect an even larger T gate count when using these techniques. The multiplication benchmark (*fp_mult*) was only synthesized using [137].

	Direct map		[98]		[137]			Our method	
name	qubits	T	qubits	T	qubits	T	v	qubits	T
fp_add					156	33521	10	49	1.44×10^9
fp_cmp					40	30426	8	37	18296672
fp_div					144	589721	12	49	2.84×10^9
fp_exp	32	784887	33	752558	32	1193083	9	33	344603
fp_invsqrt	32	136023	33	129692	32	169282	7	33	101204
fp_ln	32	969757	33	931516	32	1623461	7	33	585468
fp_log2	32	623948	33	566315	32	850331	7	33	400281
fp_mult					267	141657			
fp_recip	32	170245	33	150448	32	198167	4	33	136137
fp_sincos	33	376733	34	326403	34	452129	6	34	191136
fp_sqrt	32	122793	33	106216	32	133489	2	33	58184
fp_square	32	129797	33	126803	32	165545	5	33	108935
fp_sub					156	33626	9	49	1.47×10^9

Table 4.3: Synthesis results in number of qubits and T gates.

4.7. SUMMARY

Various quantum algorithms use subroutines that perform a classical computation on a superposition of exponentially many input states. Some examples include:

- Modular exponentiation for factoring [128].
- Evaluating orbital functions for quantum chemistry [20].
- Reciprocals for solving systems of linear equations [70]

Naturally, these applications, and other scientific computing ones, require the ability to do arithmetic using fractional (non-integer) numbers, which we can represent using either fixed-point or floating-point representations. The latter offers significant savings in the number of qubits when the required range of values and (or) relative precision are large. Thus, finding good circuits for floating-point arithmetic could be of tremendous use in many quantum computing applications.

This chapter presented a new collapsing method that outperforms existing state-of-the-art ones. Its main advantages are the improved scalability and effectiveness in collapsing all half-precision (16 bits) floating-point arithmetic benchmarks, an important step towards implementing practical algorithms for quantum computing. Also, design flows for synthesizing reversible logic in quantum computers, such as [137], can use these results as a starting point to map these operations into circuits with fewer qubits.

We also demonstrated how to apply this new collapsing method to ESOP-based reversible logic synthesis by implementing a new technique that allows us to find quantum circuits with up to 50% reduced quantum gate costs.

Chapter 5

XAG-based synthesis

5.1. MOTIVATION

The previous chapter introduced a technique capable of scaling-up ESOP-based synthesis to work with large Boolean functions. Using it allows us to synthesize quantum circuits with an optimal number of qubits, however, at the cost of significantly increasing the number of gates, as illustrated by the experimental results. In general, there exists a trade-off between the number of qubits and the number of gates in quantum circuit synthesis.

Recall that we could mitigate this gate explosion slightly by noting that we can cluster product terms with the same factor and use an additional helper qubit to store these factor values. If we take this principle up a notch, we enter the realm of hierarchical synthesis methods. In hierarchical synthesis, we decompose a Boolean function f into smaller ones, easily synthesizable by a direct method such as ESOP-based synthesis. When generating a circuit for the original f, we synthesize these small functions separately, storing their outputs in ancillary qubits.

Figure 5.1 illustrates a method known as k-LUT-based synthesis. This method starts from any logic network and decomposes it using k-LUT mapping [44] as its first step. This initial step has a relevant impact on the synthesis result. The k parameter allows us to control the maximum number of inputs for each subfunction and thus somewhat manage the number of qubits in the resulting circuit. A small k yields a decomposition of the initial function into a more significant number of subfunctions, which, in turn, leads to the synthesis of circuits with more qubits since we need qubits to store these intermediate results.

This chapter investigates improvements to a hierarchical synthesis algorithm specifically designed to work on XAGs. This technique synthesizes circuits over the Clifford+T gate set and focuses on minimizing, not only the total number of gates, but, more specifically, the number of T gates—a typical concern when dealing with fault-tolerant quantum computing.

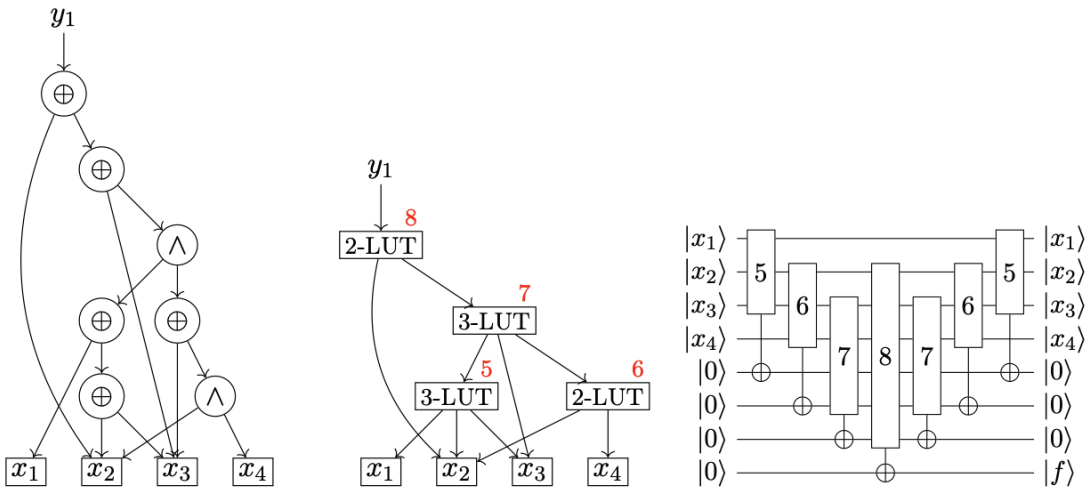


Figure 5.1: k-LUT-based synthesis: Starting from a logic network that implements the $\text{prime}_4(x) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$, we first decomposes it using k-LUT mapping [44], in this case $k = 3$.

5.2. CONTRIBUTIONS

The contributions of this chapter are as follows:

- We formally introduce a new multi-level Boolean network, high-level XAG, which allows us to explore more optimizations opportunities during synthesis.
- We present results demonstrating that our flow is capable of reducing the number of qubits by up to 24.95% and the number of Clifford gates by up to 43.3% when compared to [94]. Crucially, these improvements are possible without increasing the number of T gates nor the execution time.
- We generalize this technique to properly handle the general case of oracle synthesis $U_f : |x\rangle |y\rangle |0\rangle^k \mapsto |x\rangle |y \oplus f(x)\rangle |0\rangle^k$, i.e., when the state of the target qubits is $|y\rangle$ instead of $|0\rangle$. Note that the original implementation of [94] does not handle this case correctly.

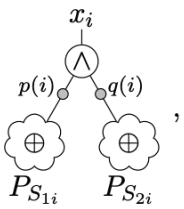
This work appears in [120] and was presented at the Design Automation & Test in Europe Conference (DATE'2020). An implementation appears in `tweedledum`.

5.3. PREVIOUS WORK

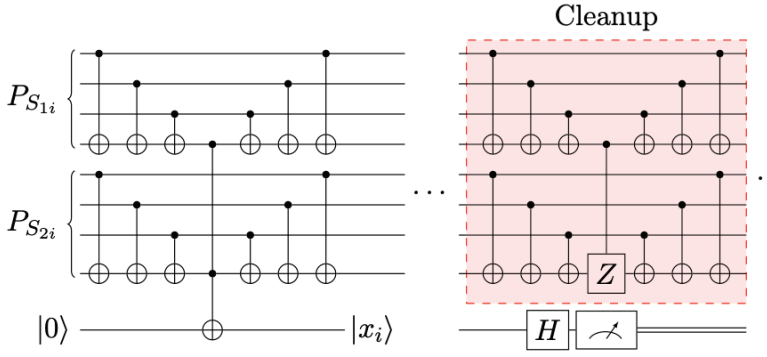
The previous state-of-the-art technique considers logic networks over the gate basis $\{\neg, \oplus, \wedge\}$ [94]. The principle behind this technique is to identify patterns in the XAG network, which conveniently translate into quantum circuits characterized by few gates. In particular, the method isolates the graph parts that translate to a single Toffoli gate, i.e., the AND gates. Doing so allows us to draw a direct correlation between the network's number of AND gates and the cost in the number of T gates and qubits.

The first step in this technique is to represent the function as a XAG and then minimize its number of AND gates. The minimum number of AND gates required to implement a Boolean function as a XAG is known as the function's *multiplicative complexity* [151], and it directly correlates to the function's resistance against algebraic attacks [47]. I called it *functional* multiplicative complexity to differentiate from the number of AND gates used in XAG representation, known as *structural* multiplicative complexity [67, 147]. These metrics play an essential role in cryptography [10, 36, 65, 112], being used to assess a function's vulnerability to attacks (potentially from quantum computers). Perhaps ironically, we can use their multiplicative minimization techniques [147, 148] to build better quantum circuits.

The key idea in reference [94, Algorithm 1] is to look at the two inputs of an AND gate as parity functions over variables that are either primary inputs or preceding AND steps:



where $P_{S_{1i}}$ and $P_{S_{2i}}$ are the parity functions over the set of variables indexed by S_{1i}, $S_{2i} \subseteq [i-1]$. Since the parity function is reversible, the inputs can be easily computed in-place, i.e., without the need for an extra qubit. The AND vertices, on the other hand, need out-of-place computation:



Note the use of a clean ancilla qubit to hold the result. The above construction was introduced in [66]. It combines ideas from [22, 105] and [75] to minimize the number of T gates. The idea is that when we are computing an AND gate that will be later cleaned up (uncomputed), we can save T gates by introducing phase errors. These errors are not a problem as long as the other intermediate operations relying on the AND are not sensitive to these errors, and we fix them during cleanup. In the construction above, during the cleanup, we execute the gates in the highlighted box only when the measurement yields 1 as a result—an unfavorable result that requires us to correct the phase.

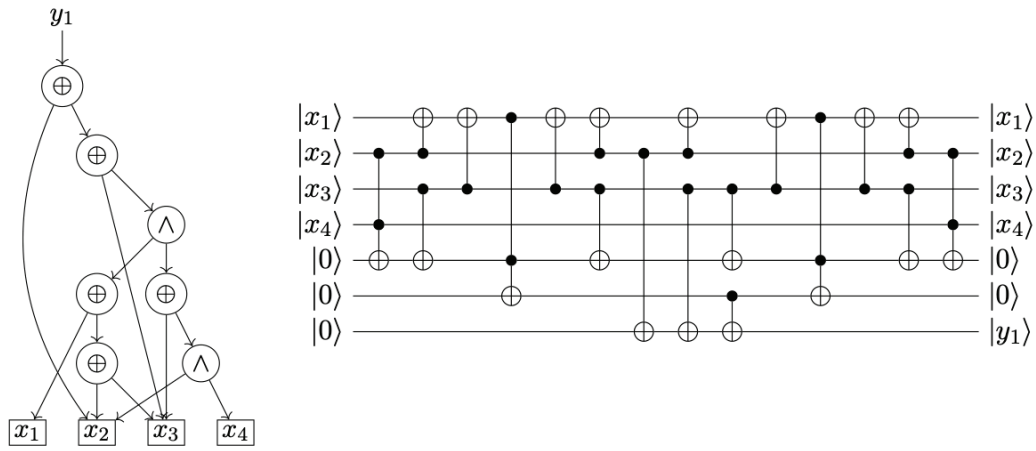


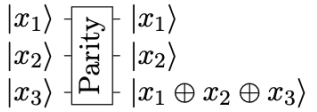
Figure 5.2: Starting from a XAG (left) that represents the $\text{prime}_4(x)$ function, we synthesize a quantum circuit (right) using [94, Algorithm 1].

The algorithm starts by computing each AND gate in three simple steps: First, it computes the input parity functions in-place, then the AND gate in a new ancilla qubit, and then it cleans up the input. After that, it copies the state of the qubit holding the output step to the output qubit. Finally, it restores the ancilla qubits to $|0\rangle$ by cleaning up all AND steps. The synthesized circuit achieves the upper bound in the number of T gates of four times the structural multiplicative complexity of the input XAG.

While we can rely on classical logic synthesis techniques to optimize the structural multiplicative complexity of the initial XAG, this problem is known to be intractable [61], and thus a hard limit on the number of ancillae may prohibit the use of this technique. One option is to use SAT to find the best strategy to compute and cleanup the required intermediate results using a target number of qubits a. Such techniques are called pebbling strategies [96] and enable the exploration of the trade-off between qubits and gates.

5.4. SYNTHESIS FLOW

Our synthesis flow uses a higher-level representation of the XAG, known as high-level XAG, that enables us to identify optimization opportunities to save qubits and Clifford gates easily. Also, our flow lowers the level of abstraction in small steps: Starting from a high-level XAG, we synthesize a reversible circuit with parity gates and Toffoli gates, where a parity gate is



Then we translate it into a circuit composed of CNOT and Toffoli gate, which, in turn, is lowered to the Clifford+T gate set. This progressive lowering allows our flow to discover more facts about the program, thus finding better optimization opportunities. For

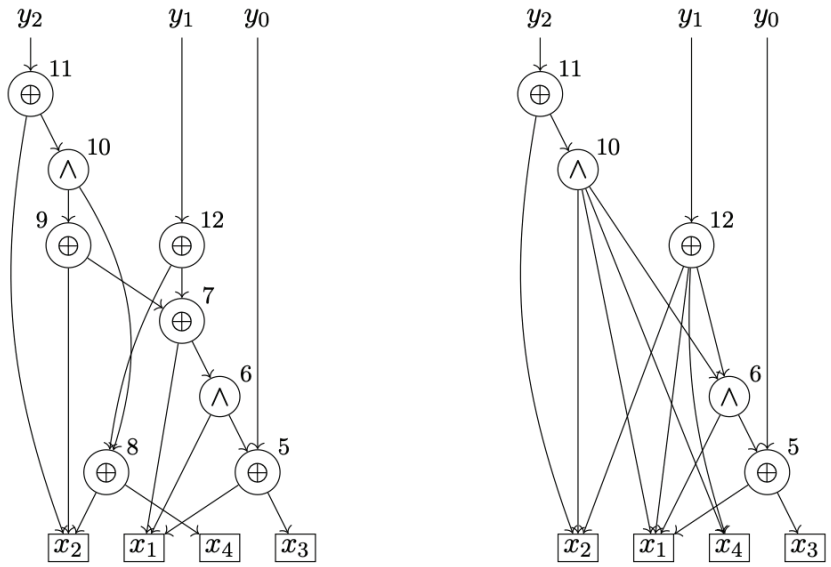


Figure 5.3: XAG and high-level XAG representations of a 2-bit ripple-carry adder. The label i outside each vertex identifies the gate x_i. In this figure, we opted to leave the labels of corresponding vertices on both graphs the same for clarity.

example, we can easily merge the parity gates and use known linear synthesis techniques such as [108] to optimal CNOT circuits that implement them.

High-level XAG. A high-level XAG is a logic network in which each local function F_{S_i} is either a 2-input AND or a 2-input XOR. The inputs, however, are parity functions, i.e. each gate has one of the two following forms:

$$x_i = P_{S_{1i}} \wedge P_{S_{2i}} \quad \text{or} \quad x_i = P_{S_{1i}} \oplus P_{S_{2i}} \tag{5.1}$$

for $n < i \leq n + r$, where $P_{S_{1i}}$ and $P_{S_{2i}}$ are parity functions over the set of variables indexed by $S_{1i}, S_{2i} \subseteq [i-1]$. Note that we can represent XOR gates as one parity function P_S. Also, we can merge XOR gates into the inputs of the AND steps, except when they drive an output. The logic level of a primary input or gate i is defined as the earliest possible time in which it can be computed. Formally,

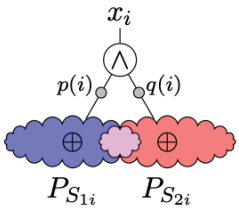
$$l_i = \begin{cases} 0 & \text{if } i \leq n, \\ \max\{l_j : j \in (S_{1i} \cup S_{2i})\} + 1 & \text{otherwise.} \end{cases}$$

Similarly, we define the reverse logic level l_i^r as the latest possible time in which gate i must be computed while not increasing the depth of the logic network.

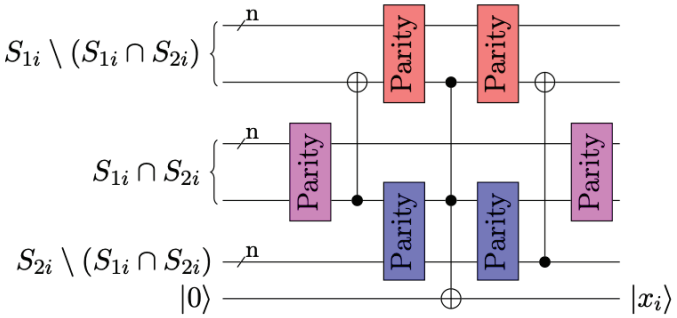
Translating a XAG into an high-level XAG is straightforward. We merge all XOR gates that are not driving an output into the input parity functions of the AND gates or the input parity function of a output XOR gate.

Like when synthesizing a circuit from a XAG, the algorithm that uses a high-level

XAG also focuses on steps that need out-of-place computation, i.e., the outputs and the AND steps. However, by using a high level of abstraction, we can simplify our algorithm implementation and more easily discover optimization opportunities. In a high-level XAG, we explicitly represent the intersection between the two parity functions that are inputs to each AND gate:



When computing the AND vertices, we start with the intersection between the two inputs:



Synthesis algorithm. In the algorithms below, each gate x_i for $0 \leq i \leq n + r$ has attributes $\texttt{LR}(i)$, $\texttt{L}(i)$, $\texttt{R}(i)$, $\texttt{FANIN}(i)$, $\texttt{REF}(i)$, $\texttt{LAST_REF}(i)$, $\texttt{LVL}(i)$, and $\texttt{TGT}(i)$, which interact as follows: $\texttt{LR}(i)$ is the set of indexes of the variables that are common in both input parity functions, i.e., $\texttt{LR}(i) = S_{1i} \cap S_{2i}$. $\texttt{L}(i)$ and $\texttt{R}(i)$ are the set of indexes of the variables that appear in only one of the input parity functions, i.e., $\texttt{L}(i) = S_{1i} \setminus \texttt{LR}(i)$ and $\texttt{L}(i) = S_{2i} \setminus \texttt{LR}(i)$. The $\texttt{FANIN}(i)$ is a vector of indexes such that $\texttt{FANIN}(i) = S_{1i} \cup S_{2i}$

The $\texttt{REF}(i)$ attribute tells the algorithm the number of references to gate x_i, including cleaning up. For example, given a gate x_k that uses x_i in its input, if x_k needs clean up, then two references are added to x_i: one for the computation of x_k and one for its clean up. However, if x_k does not need cleanup (e.g., it drives an output), then just one reference is added. $\texttt{LAST_REF}(i)$ tells the index of the last reference.

Finally, $\texttt{LVL}(i)$ is simply the reverse logic level l_i^r, and $\texttt{TGT}(i)$ indicates the target qubit of a gate. Initially, $\texttt{TGT}(i)$ is $\varnothing$ for all gates.

The synthesis process begins by assigning qubits to the primary inputs and primary outputs. The assignment is trivial for the inputs but more complicated for the outputs. For the sake of clarity, we look at the assignment as a separate algorithm.

Before delving into the details of the assignment algorithm, however, let's understand what are the complications with the primary outputs. Suppose we are given a high-level XAG with m outputs. An output can be:

1. The constant x_0.

2. A primary input.
3. The output of a AND gate.
4. The output of a XOR gate.

There are also the cases where an output is the complement of one of those, and when we have one gate driving more than one output. The first three cases are simple to handle:

1. We either do nothing or add a NOT gate to the output qubit. The later happens when the output is the complemented constant gate.
2. We use a CNOT gate to copy the classical state from the input qubit to the output qubit.
3. We compute the AND gate directly on the output qubit—saving a qubit and the gates that would be necessary to clean up.

Careful handling the fourth case allows us to save qubits and some gates. We compute the XOR gate on the output qubit. Note, however, that all explicit XOR steps are outputs. Thus, this direct computation saves us nothing. To save resources, we look among the inputs of the XOR for AND gates that we can compute on the output qubit. We are conservative in our search: The AND gate should either have only one reference or be the only AND gate among the inputs. If the AND gate has more than one reference, then we need to guarantee computing the XOR gate only after all other steps that depend on the AND.

We now present the algorithm to assign qubits to the inputs and outputs:

Algorithm A (*High-level XAG preprocessing*). Given a high-level XAG with n inputs and m outputs, a quantum circuit, and a set of $n+m$ qubits $\mathtt{Q}$, this algorithm assigns target qubits to the primary inputs and primary outputs. The set $\mathtt{Q}$ is implemented as a vector where the first n elements identify the qubits used as primary inputs and the last m the primary outputs.

A1. [*Assign PI.*] Set the target qubits of all primary input steps, that is, $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[i-1]$, for $1 \leq i < n$. (Remember the 0th step is the constant.)

A2. [*Assign AND PO.*] For each primary output i, if $\mathtt{TGT}(\mathtt{i}) = \varnothing$ and $\circ_i = \wedge$, then set $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[n + (i-1)]$.

A3. [*Assign XOR PO.*] For each primary output i in which $\circ_i = \oplus$ and $\mathtt{TGT}(i) = \varnothing$. Set $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[i-1]$ and look for a AND step among its inputs: First, we create vector $\mathtt{A}$ of all the AND that have not being assigned a qubit among the inputs of i, i.e., for $j \in \mathtt{FANIN}(i)$, if $\circ_j = \wedge$ and $\mathtt{TGT}(\mathtt{j}) = \varnothing$, then add j to $\mathtt{A}$. Now, for $k \in \mathtt{A}$, if $\mathtt{REF}(i) = 1$ or $\mathtt{SIZE}(A) = 1$, and $\mathtt{LVL}(\mathtt{LAST_REF}(k)) \leq \mathtt{LVL}(i)$; then set $\mathtt{TGT}(k) \leftarrow \mathtt{TGT}(i)$, $\mathtt{REF}(k) \leftarrow \mathtt{REF}(k) - 1$.

In the following algorithm, we use functions that have the same name as quantum gates to denote the addition of the respective gate to the quantum circuit. The functions TOFFOLI and PARITY take two inputs: A set of controls qubits and a target qubit.

Algorithm B (*Synthesize circuit*). Given a high-level XAG with n inputs and m outputs, a quantum circuit, and a set of $n+m$ qubits $\mathtt{Q}$, this algorithm assign target qubits to the primary inputs and primary outputs.

B1. [*Preprocessing.*] Execute Algorithm A.

B2. [*Compute level.*] We process the nodes of the high-level XAG level by level. We set $l_{\max} \leftarrow \max\{l_i \in 0 \leq i < n\}$. For $1 \leq h < l_{\max}$, we do: for all nodes i such that $\mathtt{LVL}(i) = h$, if i is an XOR node, i.e., $\circ_i = \oplus$, execute step B3; Otherwise execute step B4.

B3. [*Compute XOR.*] We directly compute the gate on its target qubit: PARITY($\{\mathtt{TGT}(j) : j \in \mathtt{FANIN}(i), \mathtt{TGT}(j) \neq \mathtt{TGT}(i)\}, \mathtt{TGT}(i)$). Then we update the reference counters of all nodes used by i: $\mathtt{REF}(j) = \mathtt{REF}(j) - 1$ for $j \in \mathtt{FANIN}(i)$.

B4. [*Compute AND.*] If $\mathtt{TGT}(i) = \varnothing$, we request an ancilla a and set $\mathtt{TGT}(i) \leftarrow a$. If $|\mathtt{L}(i)| < |\mathtt{R}(i)|$, we swap them $\mathtt{L}(i) \leftrightarrow \mathtt{R}(i)$. We choose two qubits k and v to hold the in-place computation of the inputs. We set k to be an element of $\mathtt{L}(i)$. If $\mathtt{LR}(i)$ is not empty, then we choose $v \in \mathtt{LR}(i)$; Otherwise, we pick $v \in \mathtt{R}(i)$. We execute step B3 and add a Toffoli: TOFFOLI($\{\mathtt{TGT}(k), \mathtt{TGT}(v)\}, \mathtt{TGT}(i)$. We also update the reference counters of all nodes used by i: $\mathtt{REF}(j) = \mathtt{REF}(j) - 1$ for $j \in \mathtt{FANIN}(i)$. Lastly, we clean up the inputs by executing B5 in reverse oder.

B5. [*Compute outputs.*] Do:

- PARITY($\{\mathtt{TGT}(j) : j \in \mathtt{L}(i), j \neq k\}, \mathtt{TGT}(k)$)
- PARITY($\{\mathtt{TGT}(j) : j \in \mathtt{LR}(i), j \neq v\}, \mathtt{TGT}(v)$)
- If $\mathtt{LR}(i)$ is not empty: PARITY($\{\mathtt{TGT}(v)\}, \mathtt{TGT}(k)$)
- PARITY($\{\mathtt{TGT}(j) : j \in \mathtt{R}(i), j \neq v\}, \mathtt{TGT}(v)$)

B6. [*Recursively try to cleanup.*] For $j \in \mathtt{FANIN}(i)$, if $\mathtt{REF}(i) = 0$, then we compute j using step B3 and try to cleanup the inputs j using this step.

B7. [*Missing outputs.*] This step takes care of any primary output that might not have been computed. For example, a primary output that is a primary input. Set $i = n$. For each primary output o: We do PARITY($\{\mathtt{TGT}(i)\}\mathtt{Q}[i]$) if $\mathtt{Q}[i] \neq \mathtt{TGT}(o)$ and $o \neq 0$, and then $i \leftarrow i + 1$.

B8. [*Complement outputs.*] Set $i = n$. For each primary output: We do X($\mathtt{Q}[i]$) if the output is complemented, and then $i \leftarrow i + 1$. ∎

These algorithms are able to handle the synthesis of both forms, i.e., the general form given $U_f : |x\rangle\,|y\rangle\,|0\rangle^k \mapsto |x\rangle\,|y \oplus f(x)\rangle\,|0\rangle^k$, and the zero-target form where $|y\rangle$ equals $|0\rangle$. When computing the zero-target form, however, we can get better results by adapting these algorithm to take advantage of the fact that we know the output qubits are on state $|0\rangle$. In such case, we try to postpone the computation of the outputs as much as possible, so that we can use those qubits as ancilla for the intermediate computations.

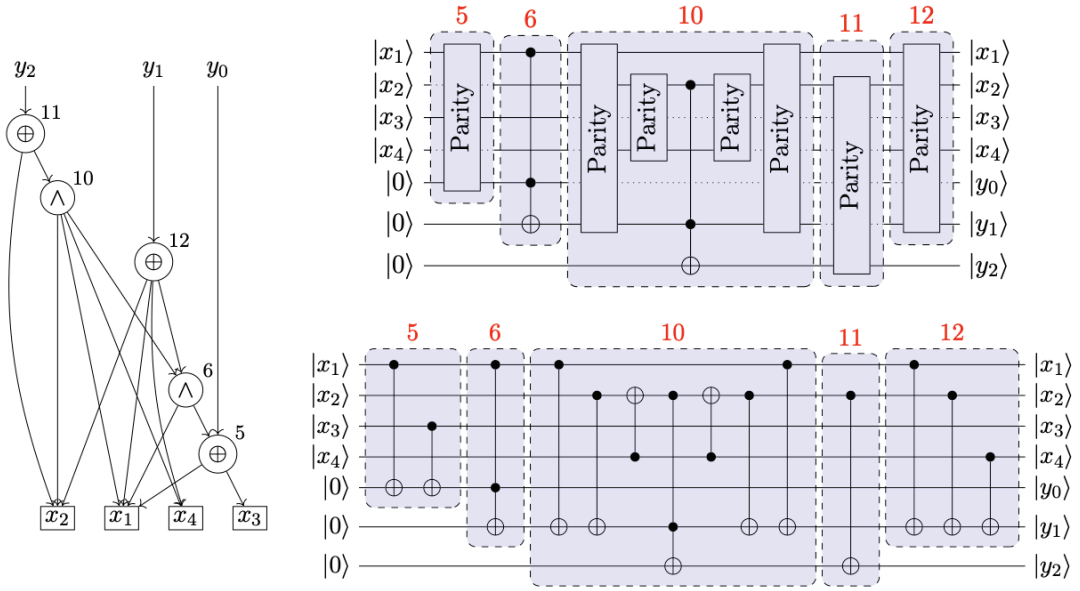


Figure 5.4: Starting from the XAG in Figure 5.2, we first create a high-level XAG (left). We synthesize a quantum circuit with parity gates and Toffoli gates (top right); Then we lower it to a circuit composed of CNOT gates and Toffoli (bottom right), which, in turn, would be lowered to a circuit over the Clifford+T gate set.

Also note that, for simplicity, we chose to not show measurement-based cleanup in Algorithm B. It is, however, straightforward to adapt the algorithm to include this technique: We just need to add slightly modified versions of steps B4 and B5 to cleanup AND gates. Our C++ implementation incorporate such changes.

5.5. EXPERIMENTAL RESULTS

We use various arithmetic and random-control functions from [14] as well as cryptographic functions and IEEE floating-point operations [7] as benchmarks. For the EPFL benchmarks, we list the currently best-known results for multiplicative complexity obtained from the state-of-the-art optimization approaches in [94]. We compile the functions into quantum operations of two forms: the general form and the zero-target form. The resulting quantum circuit representation uses the Clifford+T gate set.

In [94], the authors provide a C++ implementation of their algorithm that does not properly handle the general form. The problem lies with the measurement-based cleanup. Their implementation might apply this technique on Toffoli gates acting on the output qubits, thus destroying the initial state $|y\rangle$. We modify their implementation to correct this behavior. For the first form, we compare our implementation against the modified version of their implementation. We can directly compare the results of synthesizing into the zero-target form.

We report the results in Table 5.1. The difference between synthesizing into the different forms is only on the number of qubits; Both T and Clifford gate counts stay

the same. Thus, we only distinguish between the two forms in the qubits column: The results in parentheses correspond to the zero-target form. Note that we don't ignore any Clifford gates. (In [94], the Clifford gates in the decomposition of Toffoli gates to a Clifford+T gate set are ignored.) Also, we report the number of Clifford gates for the worst case, i.e., we assume that all measurement-based cleanups fail—meaning that we will need Clifford to fix the phase.

EPFL benchmarks On these benchmarks our algorithm improves, on average, 4.07% the number of qubits and 8.72% the number of Clifford gates. The largest improvement is of 24.95% in the number of qubits and 43% in the number of Clifford gates, and occurs for the adder, when synthesizing for the general form. Such impressive gain is due to Algorithm A, which is able to correctly identify that all Toffoli gates can be directly computed on the output qubits, and, thus, don't need to be cleanup.

Cryptographic functions & IEEE floating-point We were unable to obtain results for all crypto benchmarks due to memory constraints. The state-of-the-art fails on *Keccak-f*, *SHA-256* and *SHA-512*. Our method, on the other handle, can handle *SHA-256* because of its multilevel intermediate representation, which is more memory efficient. The other results show a significant improvement on the number of Clifford gates: 22.21% on average. On the IEEE floating-point operations we also observe an important gain in the number of Clifford gates.

5.6. SUMMARY

In this chapter, we presented a flow for oracle synthesis. The flow allows quantum algorithm designers to define the classical behavior of these quantum operations on a high level of abstraction, e.g., a Python function. Then we translate it to a XAG that we optimize using classical logic synthesis techniques. We focused on XAGs with a minimal number of AND gates since this number is proportional to the T-count of the resulting quantum circuit. The impact of the number of XOR nodes in the graph, on the other hand, should be better studied in the future. In our experiments, we have seen that the relation between the number of XOR steps and the number of CNOT gates is not so straightforward.

Our technique achieves better results compared to other state-of-the-art synthesizers. We can reduce the number of qubits by 24.95% and the number of Clifford by 43.3% in the best cases. Crucially, these improvements were possible without increasing the number of T gates or the execution time.

In addition, our multilevel intermediate representation (IR) allowed us to manipulate circuits with a more significant number of gates more efficiently. For example, we dealt with a benchmark that required too much memory when represented using a low-level IR. Also, we can more easily identify the linear parts of the quantum circuit to apply resynthesis technique to reduce the CNOT overhead.

Benchmark	State-of-the-art [94]				Proposed				Gain		
	qubits		T	Clifford	qubits		T	Clifford	qubits		Clifford
Arithmetic functions [14]											
adder	513	(385)	512	3820	385	(385)	512	2167	24.95%	—	43.3%
bar	1095	(1032)	3328	33536	1095	(1095)	3328	32768	—	(-6.10%)	2.29%
div	6252	(6188)	24240	333081	5950	(5948)	24240	306211	4.83%	(3.88%)	8.07%
log2	19500	(19469)	77744	1201139	19479	(19458)	77744	1118969	0.11%	0.06%	6.84%
max	1572	(1444)	3724	21854	1443	(1315)	3724	19289	8.21%	8.93%	11.74%
multiplier	12194	(12069)	47760	1017983	12169	(12065)	47760	686236	0.21%	0.03%	32.59%
sin	4122	(4099)	16300	112934	4098	(4095)	16300	108793	0.58%	0.10%	3.67%
sqrt	6376	(6372)	24976	283381	5961	(5960)	24976	245303	6.51%	6.47%	13.44%
square	5365	(5246)	20724	309490	5312	(5235)	20724	259974	0.99%	0.21%	16.00%
Random control [14]											
arbiter	1437	(1437)	4724	13657	1437	(1437)	4724	13657	—	—	—
cavlc	505	(504)	1976	6506	497	(487)	1976	6488	1.58%	3.37%	0.28%
ctrl	93	(93)	340	943	90	(87)	340	943	3.23%	6.45%	—
dec	349	(349)	1364	3068	349	(349)	1364	3068	—	—	—
i2c	777	(771)	2492	8285	758	(739)	2492	8193	2.45%	4.15%	1.11%
int2float	113	(111)	400	1531	103	(98)	400	1494	8.85%	11.71%	2.42%
mem	6881	(6319)	20452	72535	6241	(5655)	20452	68521	9.30%	10.51%	5.53%
priority	458	(455)	1308	4537	451	(447)	1308	4518	1.53%	1.76%	0.42%
voter	6652	(6652)	22604	237227	6652	(6652)	22604	215023	—	—	9.36%
Crypto functions [7]											
AES-128	6784	(6659)	25600	2947672	6752	(6752)	25600	2288312	0.47%	(-1.40%)	22.37%
AES-192	7616	(7491)	28672	3292468	7584	(7584)	28672	2560868	0.42%	(-1.24%)	22.22%
AES-256	9344	(9219)	35328	3690293	9312	(9312)	35328	2877317	0.34%	(-1.01%)	22.03%
Keccak-f[†]											
SHA-256[†]					23585	(23585)	90292	583182318			
SHA-512[†]											
IEEE floating-point [7]											
FP-add	5576	(5513)	21536	600804	5512	(5512)	21536	536401	1.15%	(0.02%)	10.72%
FP-div	82457	(82394)	329060	36311574	82393	(82393)	329060	32636047	0.08%	—	10.12%
FP-eq	506	(506)	1260	4094	506	(443)	1260	4032	—	(12.45%)	1.51%
FP-f2i	1594	(1531)	5868	38416	1531	(1529)	5868	38070	3.95%	(0.13%)	0.90%
FP-mul	19806	(19743)	78456	2217382	19742	(19742)	78456	2002297	0.32%	(0.01%)	9.70%
FP-sqrt	91520	(91457)	365568	84586674	91456	(91456)	365568	73925221	0.07%	—	12.60%

[†] The missing results are due to the system running out of memory (OOM).

Table 5.1: Experimental results

Chapter 6

Symbolic algorithms for permutation synthesis

6.1. MOTIVATION

In this chapter, we study symbolic algorithms to solve two related problems on graphs: the *token swapping problem*, introduced by Yamanaka et al. [155], and the *permutation routing via matchings* problem, proposed by Alon et al. [12]. In both, we are given a connected graph with n vertices, a set of n tokens, and an initial bijective assignment between tokens and vertices, i.e., a permutation. We are also given a target permutation. The goal is to move each token to its destination vertex in the target permutation by applying a sequence of token swaps among adjacent vertices. We refer to both as a *reconfiguration problem.*

The difference between the two problems lies in the optimization goal. In the token swapping problem, the aim is to minimize the total swaps. In the permutation routing via matchings, the goal is to minimize the total number of steps by picking matchings, i.e., a disjoint collection of edges, and swapping the tokens of all vertices connected by an edge on the matching at each step—some authors have referred to this as the parallel token swapping problem.

Both problems appear in the context of quantum circuit mapping [157]: the process of modifying a high-level quantum circuit, which assumes full qubit connectivity, into a lower-level one that respects the device's connectivity (or coupling) constraints. This process is not always possible without including additional gates. A SWAP-based mapper models mapping as a reconfiguration problem in which the vertices correspond to the device's physical qubits and the tokens to the high-level circuit's virtual qubits. A two-qubit gate can only be executed between virtual qubits mapped to adjacent physical qubits. The mapper uses SWAPs to move virtual qubits to adjacent physical ones when this is not the case.

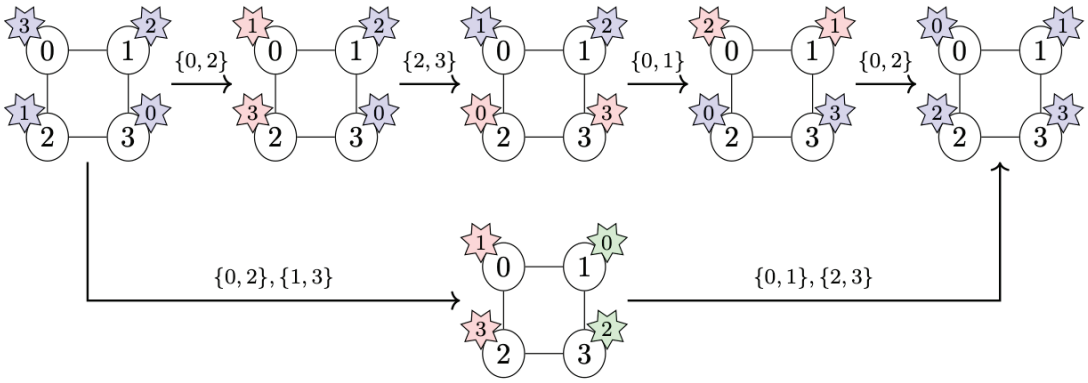


Figure 6.1: This figure shows two solutions to a reconfiguration problem. Give a connected graph with a set of tokens placed on its vertices; our goal is to use swap operations to equalize the tokens' and vertices' labels. On the top solution, we employ four steps with one swap operation, which is optimal number of swaps. On the bottom solution, we use two steps with two swaps each. We can apply swap in parallel in both steps because they work on disjoint sets of vertices.

6.2. TECHNICAL BACKGROUND

In the following, we introduce the necessary technical background to understand our algorithms to solve these problems.

6.2.1 Graphs

A *graph* is an ordered pair of sets $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges or arcs. In an *undirected* graph, the edges are unordered pairs. In this work, we write $u \text{ — } v$ instead of $\{u, v\}$ to denote the undirected edge between vertices u and v. In a *directed* graph, we use the term arc in a set $E \subseteq V \times V$ to denote a link between two vertices and write $u \longrightarrow v$ instead of (u, v) to denote an arc from u to v.

For any edge $u \text{ — } v$ in an undirected graph, we call u a *neighbor* of v, and vice versa. We denote $\delta(v)$ the set of neighbors of v and the *degree* of a vertex is $|\delta(v)|$, i.e., its number of neighbors. In directed graphs, we distinguish two kinds of neighbors. For any directed edge $u \longrightarrow v$, we call u a predecessor of v and v a successor of u. Accordingly, we use $\delta^-(v)$ and $\delta^+(v)$ to denote the set of predecessors and the set of successors of a vertex v respectively.

Given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we say that a *subgraph isomorphism* from G_1 to G_2 is an injective function $f : V_1 \to V_2$ such that $v_i \text{ — } v_j$ implies $f(v_i) \text{ — } f(v_j)$ for all $v_i, v_j \in V_1$. A *matching* M of a graph G is subgraph where every vertex has degree 1, i.e., no two edges on M have a common vertex.

In a directed graph, a *directed walk* is a sequence of vertices $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_l$ such that $v_{i-1} \longrightarrow v_i \in E$ for every index i. A directed walk is called a *directed path* if it visits each vertex at most once.

6.2.2 Permutation

A permutation is a bijective function $f : X \to X$ where X is a finite set $\{1, 2, \ldots, n\}$. A permutation is represented as a tuple $\pi = (a_1, a_2, \ldots, a_n)$ in which each item k moves to a_k. For example, $\pi = (3, 1, 2, 4)$ implies $1 \mapsto 3$, $2 \mapsto 1$ and $3 \mapsto 2$—in this example item 4 is already on its place. We write S_n to denote the symmetric group on X, i.e., the set of all permutations, and use π_e to denote the identity permutation $(1, 2, \ldots, n)$. A transposition $\tau_{(i,j)} \in S_n$ is an elementary permutation which interchanges two elements and leaves all others unchanged. Any permutation can be decomposed into a sequence of transpositions which can be made canonical. We can represent the previous example permutation as $\pi = \tau_{(3,1)} \circ \tau_{(2,1)}$. Note that we apply transpositions from right to left.

An *inversion* of π is a pair of indices $i < j$ such that $\pi(i) > \pi(j)$, where $i, j \in [n]$ and $\pi(i)$ denotes the i^{th} element. The number of inversions of π is denoted $\text{inv}(\pi)$. The *sign* of a π is $\text{sgn}(\pi) = (-1)^{\text{inv}(\pi)}$. For any transposition $\text{sgn}(\tau_{(\text{i,j})}) = -1$. For all $\pi_i, \pi_j \in S_n$, $\text{sgn}(\pi_i \circ \pi_j) = \text{sgn}(\pi_i) \cdot \text{sgn}(\pi_j)$.

Theorem 1. *Let $\pi = \tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)} \circ \tau_{(a_1,b_1)}$ be any decomposition of $\pi \in S_n$ into a sequence of transpositions. Then*

$$\text{sgn}(\pi) = (-1)^k.$$

Proof. We can rewrite π as $\pi_2 \circ \pi_1$ where $\pi_1 = \tau_{(a_1,b_1)}$ and $\pi_2 = \tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)}$, then $\text{sgn}(\pi) = \text{sgn}(\pi_1) \cdot \text{sgn}(\pi_2)$. Next we can do the same for π_2 and iteratively for all subsequent π_k such that each π_k is a transposition$\tau_{(a_k,b_k)}$. Clearly, $\text{sgn}(\pi) = \prod_{i=1}^{k} sgn(\pi_i)$. The conclusion now follows by recalling that $\text{sgn}(\tau_{(\text{i,j})}) = -1$. □

6.2.3 Reconfiguration problems

Definition 1 (Token assignment)**.** *Given an undirected graph $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$, a* token assignment *is a bijective mapping $\Pi : V \to \{1, \ldots, n\}$. By assuming an order on the vertices $v_1 < \cdots < v_n$, a token assignment can equally be described by a permutation $\pi \in S_n$, where $\pi(i) = \Pi(v_i)$.*

Given a connected undirected graph $G = (V, E)$, an initial token assignment $\pi_{\text{init}} = \pi_0$ and a target token assignment π_{target}. We define the two problem of interest as follows.

Problem 1 (Token swapping, [155])**.** *Find a sequence of transpositions $\tau_{(a_1,b_1)}, \ldots, \tau_{(a_k,b_k)}$ with v_{a_i} — v_{b_i} such that*

$$\pi_i = \pi_{i-1} \circ \tau_{(a_i,b_i)} \qquad \text{for all } 1 \leq i \leq k,$$

and $\pi_k = \pi_{\text{target}}$, where k is minimum.

Problem 2 (Permutation routing via matchings, [12])**.** *Find a sequence of matchings*

$M_1, \ldots, M_\ell \subseteq E$ on G such that

$$\pi_i = \pi_{i-1} \circ \prod_{\{v_{a_i}, v_{b_i}\} \in M_i} \tau_{(a_i, b_i)} \qquad \text{for all } 1 \leq i \leq \ell,$$

and $\pi_\ell = \pi_{\text{target}}$, where ℓ is minimum.

Note that it is always possible to relabel the graph such that π_0 is the identity permutation, $\pi_0 = \pi_e$. Hence, we assume such an initial token assignment in the remainder of this chapter without loss of generality.

6.2.4 State space

A state-space is a 6-tuple $\mathcal{S} = \langle S, A, \text{cost}, T, s_0, S_\star \rangle$ where S is a finite set of states, A is a finite set of actions, $\text{cost} : A \to \mathbb{R}_0^+$, $T \subseteq S \times A \times S$ is a transition relation (deterministic in $\langle s, a \rangle$), $s_0 \in S$ is the initial state, and $S_\star \subseteq S$ is the set of goal states. State-spaces are often represented as directed graphs. Given a state-space $\mathcal{S}$, we can construct a directed graph $D_s = (S, E)$, where the set of vertices V consists of all possible states, and an arc $v \longrightarrow w$ only if there exists an action $a \in A$ that transforms the state in v to the state in w, denoted $v \underset{a}{\to} w$.

6.2.5 Decision diagrams

Zero-suppressed decision diagram (ZDD, [100]). A ZDD is a graph-based representation of a finite family of finite subsets. Given a set of variables $X = x_1, \ldots, x_n$, a ZDD is a directed acyclic graph with non-terminal vertices N, also called decision vertices, and two terminal vertices $\top$ and $\bot$. Each vertex $v \in N$ is associated with variable $V(v) \in X$ and has two successor vertices $\text{HI}(v), \text{LO}(v) \in N \cup \{\top, \bot\}$. The vertices on a directed path to a terminal vertex follows a fixed variable order which guarantees the canonicity of the ZDD.

Each vertex in the ZDD represents a finite family of finite subsets over X. The terminal node $\bot$ represents the empty family $\emptyset$ and the terminal vertex $\top$ represents the unit family—a family containing the empty set $\{\emptyset\}$. Each decision vertex v represents the subset

$$\text{LO}(v) \cup \{S \cup \{x_{V(v)}\} \mid S \in \text{HI}(v)\}.$$

Given two ZDDs f and g, the following list of operations is part of what is called a ZDD family algebra. Each operation can be efficiently implemented using ZDDs.

union	$f \cup g = \{\alpha \mid \alpha \in f \text{ or } \alpha \in g\}$
intersection	$f \cap g = \{\alpha \mid \alpha \in f \text{ and } \alpha \in g\}$
difference	$f \setminus g = \{\alpha \mid \alpha \in f \text{ and } \alpha \notin g\}$
nonsupersets	$f \searrow g = \{\alpha \in f \mid \beta \in g \text{ implies } \alpha \not\supseteq \beta\}$

For a detailed description of how ZDDs are represented in memory and how the family algebra operations are implemented, we refer the reader to the literature [80, 100].

Permutation decision diagram (πDD, [101]). A πDD is a ZDD-like data structure for representing finite sets of permutations. Minimal changes to ZDD semantics allows πDD to be build on top of a ZDD implementation. First, each transposition is represented by a variable in a ZDD.

Each vertex in the πDD represents a finite set of permutations. The terminal vertex $\bot$ retains its semantics, i.e., it represents an empty set of permutation $\emptyset$, while $\top$ represent the unit set which is the set containing identity permutation π_e. Each decision vertex represents the subset

$$\mathrm{LO}(v) \cup \{\mathrm{HI}(v) \cdot \tau_{(x,y)}\}$$

Given two πDDs f and g, the binary set operations: union, intersection and difference work in the same manner as in ZDDs. What differentiate πDD is the following two operations.

$$\begin{array}{ll} \textit{transposition} & f \cdot \tau_{(x,y)} = \{\alpha \cdot \tau_{(x,y)} \,|\, \alpha \in f\} \\ \textit{cartesian product} & f * g = \{\alpha\beta \,|\, \alpha \in f,\, \beta \in g\} \end{array}$$

6.3. CONTRIBUTIONS

In this chapter, we present the results of research directed towards the development and analysis of symbolic algorithms for solving both problems:

- *A*-based algorithm.* We study the use of A* search algorithm [71] to solve both problems using admissible heuristics, which guarantee optimality and non-admissible heuristics.
- When trying to optimize for the number of swaps, we prove that any non-admissible heuristic will obtain a result that deviates from an optimal result by an even number of swaps.
- *SAT-based algorithm.* Inspired by [144], we introduced a new encoding which relies only on SAT for solving both problems.
- Finally, we show how both problems emerge in the context of quantum circuit mapping [157].

This work appears in [124] and was presented at IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL'2020). An implementation appears in `tweedledum`.

6.4. A*-BASED ALGORITHM

We can abstract both problems to the mathematical problem of finding a minimal cost path from a start vertex to a goal vertex in a directed graph $D_s = (S, E)$ representing a state space. Each vertex $s \in S$ represents a state in such a graph, i.e., in our case, one possible token assignment. The set of actions, which transforms a state, changes

depending on the problem. In the token swapping problem, A corresponds to the set of edges in G, while in the permutation routing via matchings, A corresponds to the set of matchings $\mathcal{M}$ in G.

The graph D_s is not explicitly specified as the number of vertices and edges is too large. Instead, the graph is implicitly represented by means of a set of source vertices $S' \subset S$ and a successor operator $\Gamma : S' \to (S \times \text{cost})^{|A|}$, i.e., an operator that, given a vertex s_i, generates the set of tuples $(s_k, \text{cost}(a))$, where $s_k \in \delta^+(s_i)$, for all $a \in A$.

We employ an informed search algorithm, namely A* [71], to solve the problem of finding a lowest-cost path in D_s. Given a start vertex s_0, the algorithm iteratively generates parts of a subgraph of D_s by applying the successor operator $\Gamma(s_i)$. We say that a vertex has been expanded when the successor operator is applied to it. For each expanded vertex $s_i \in D_s$ a weight is calculated for all its successors s_k, $s_i \underset{a}{\to} s_k$, using

$$\text{weight}(s_k) = g(s_k) + h(s_k),$$

where $g(s_k) = g(s_i) + \text{cost}(a)$ is the cost to reach vertex s_k and $h(s_k)$ is a heuristic function that estimates the cost from s_k to the target vertex. In other words, $\text{weight}(s_k)$ gives an estimate of the total cost of a path using that vertex. At each iteration, the vertex with the lowest cost is chosen to be expanded. The algorithm expands the vertex with the lowest cost first, some parts of the search space (those that lead to expensive solutions) are never explored. Hence use of a good heuristic is important in determining the performance of A*. Further, if $h(s_k)$ is admissible, that is, it never overestimates the cost of the cheapest path from s_k to a target vertex, then A* guarantees finding an optimal solution.

Token swapping. We describe two heuristics for solving the token swapping problem. The first is admissible and follows from the following theorem.

Theorem 2. *Let $d(\Pi(v_i))$ be the distance of a token $\Pi(v_i)$ to its target vertex v_t. Let K be the sum of distances of all tokens to their target vertices $K = \sum_{i=1}^{n} d(\Pi(v_i))$, then*

$$h(s_k) = \frac{K}{2}$$

is an admissible heuristic for solving the token swapping problem.

Proof. Any solution would need a least $\frac{K}{2}$ swap operations as every swap reduces K by at most 2. □

Experiments demonstrate that such admissible heuristic can still take a long time to find optimal solutions for problems with up to 20 vertices. Given the inherent complexity of the problem, this is a good indication that employing an admissible heuristic might be prohibitive, especially if the problem instances grow. Hence, we explored different non-admissible heuristics which aggressively prune the search space. We report results for when using $h(s_k) = K$. Furthermore, we prove the following corollary to Theorem 1.

Corollary 1. *Any non-optimal solution to the token swapping problem differs from an optimal solution by an even number of swaps.*

Proof. Let π_{target} be any permutation and $\tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)} \circ \tau_{(a_1,b_1)}$ be any sequence of swaps which transforms the identity permutation π_e into π_{target}. If $\text{sgn}(\pi_{\text{target}})$ is 1 (-1), then such sequence must have an even (odd) number of swaps, since $\text{sgn}(\pi_{\text{target}}) = (-1)^k$. The conclusion now follows because this is true for both optimal and non-optimal sequences. □

Permutation routing via matchings. Solving this variant of the problem requires knowing all the matchings of the graph, i.e., computing the set of all combinations of edges in which no two edges have a common vertex. We use ZDDs to represent the set of all matchings. Given a graph $G(E, V)$ the ZDD is defined over the $|E|$ variables $e \in E$. The ZDD with all matchings $\mathcal{M}$ is described by

$$\mathcal{M} = \wp \searrow \bigcup_{v \in V} \binom{\delta(v)}{2},$$

where $\wp$ refers to the ZDD that represents the universal family of all subsets of E.

We also employ one admissible and one non-admissible heuristic to solve this problem. The admissible heuristic follows from the following simple lemma.

Lemma 1. *Let $d(\Pi(v_i))$ be the distance of a token $\Pi(v_i)$ to the target v_i. Let K be the maximum distance of all tokens to their target vertices $L = \max d(\Pi(v_i))$, then*

$$h(s_k) = L$$

is an admissible heuristic to solve the permutation routing via matchings problem.

Proof. Any solution needs at least L steps as every step reduces L by at most 1. □

6.5. SAT-BASED ALGORITHM

We use the state-space representation to formulate the token swapping and permutation routing via matching problems as the Boolean satisfiability problem. In the encoding, I use two types of variables: one indicates whether a token is placed in a vertex at a given step, and the other indicates whether a swap between two vertices took place at a given step. Our encoding uses four different types of clauses to constrain the problem such that the solution corresponds to a correct placement of swaps:

C1. At each level, each token must be assigned to exactly one vertex and each vertex must be assigned to exactly one token.

C2. If at steps l and $l + 1$ a vertex is assigned the same token, then no swapping involving that vertex occurred at l.

C3. If at levels l and $l+1$ a vertex is assigned to different tokens, then a swap involving that vertex occurred and must have occurred with its adjacent vertex that at level l is assigned the token.

The last necessary constraint changes depending on the problem we are solving. The constraint C4a is necessary when searching for an optimal number of swaps, while constraint C4b is necessary when searching for an optimal number of matchings.

C4a. At each level at most one swap can occur.

C4b. At each level, each vertex can only be involved in at most one swap.

Let x_{tv}^l be a Boolean variable which indicates whether a token t is assigned to the vertex v at the level l. Constraint C1 can be split into two parts: one guarantees that each token is assigned to exactly one vertex, expressed as

$$\forall l \; \forall t, \sum_v x_{tv}^l = 1,$$

and the other ensures that each vertex is assigned to one token

$$\forall l \; \forall v, \sum_t x_{tv}^l = 1.$$

Let s_{vw}^l be a Boolean variable representing whether a swap between vertices v and w occurs at step l. The constraint C2

$$x_{tv}^l \wedge x_{tv}^{l+1} \Rightarrow \bigwedge_{w \in \delta(v)} \overline{s}_{vw}^l,$$

where $\delta(v)$ is the set of vertices adjacent to v. Similarly, constraint C3

$$\overline{x}_{tv}^l \wedge x_{tv}^{l+1} \Rightarrow \bigvee_{w \in \delta(v)} (s_{vw}^l \wedge x_{tw}^l).$$

Constraint C4a can be expressed as

$$\forall l \sum_{\{v,w\} \in E} s_{vw}^l \leq 1.$$

Constraint C4b allows multiple non-conflicting swaps per step, i.e., a set of swaps that act on different vertices, and is described by

$$\forall l \; \forall v, \sum_{w \in \delta(v)} s_{vw}^l \leq 1.$$

As the SAT-based approach always answers a given formula in a satisfiable/unsatisfiable (yes/no) manner we need to translate the minimization of the number of swaps

(or steps) into series of queries to the SAT solver. We build a formula that encodes a question of whether there is a solution to the problem using a specified number of swaps (or steps) using the above constraints. In practice, for both optimizations goals, this corresponds to the number of steps l we encoded.

Our implementation solves the problem incrementally by adding a new step whenever the SAT formula is unsatisfiable. We do not always start with $l = 1$ as it is possible to analyze each problem instance to get a lower bound on the necessary number of swaps (or steps) required. When optimizing the number of swaps, the lower bound corresponds to the sum of shortest paths connecting the start and target vertices of each token divided by 2 (see Lemma 2), while the lower bound on the number of steps corresponds to the longest shortest path (see Lemma 1).

The encoding can have an enormous effect on the speed with which a SAT solver can find an answer to our problem. Thus it is important to consider means to improve it. In this case, we can reduce the number of variables and simplify some clauses by noting that when encoding the possibility of a token been at a certain vertex at a given step l, we only need to consider those vertices that reachable by the token in l steps from its initial position. Further, when optimizing for the number of swaps, we can reduce the number of queries to the solver by encoding two new steps, instead of one, whenever the solver returns unsatisfiable—this can be done because of Lemma 2.

6.6. πDD-BASED ALGORITHM

As discussed in the previous section, πDDs allow for a compact representation of sets of permutations as well as efficient operations such as the Cartesian product.

The set for permutations given by using one SWAP operation is $\bigcup_{i=1}^{n-1} \tau_{(i,i)}$. Thus, all possible configurations generated by up to k SWAPs are given by

$$\begin{aligned} P_0 &= \pi_e \\ P_1 &= P_0 \cup \bigcup_{i=1}^{n-1} \tau_{(i,i)} \\ P_k &= P_{k-1} \times P_1, \text{ for } k \geq 2 \end{aligned} \tag{6.1}$$

Using this formula we can increase k until $P_{k+1} = P_k$ for any $k \geq m$. This means that m is the minimum number of SWAPs required to cover all possible configurations.

Based on the results and procedures of the previous section, a algorithm for solving the token swapping problem with a minimal number of SWAPs can be formulated. For this purpose, we first represent the final configuration as permutation π_f. Then, all permutations are enumerate as in (6.1). Ater each step k, we check whether π_f is contained in the set of all permutations P_k. When this check returns a positive answer, then the minimum number of SWAPs is k. Then, we need to extract the sequence of SWAPs. For this purpose, the algorithms moves backwards going from P_k to to P_0 by applying gates in T. The algorithms can be formulated as follows.

Algorithm E (*πDD-based synthesis*). The final permutation π_f and the set of allowed transpositions T are given.

E1. [*Initialize.*] Set $P_0 \leftarrow \{\pi_e\}$, $P_1 = P_0 \cup T$ and $k \leftarrow 1$.

E2. [*Found minimum?*] If $P_k \cap \{\pi_f\} \neq$, i.e., $\pi_f \in P_k$, go to step E4.

E3. [*Increase k.*] Set $k \leftarrow k+1$, $P_k \leftarrow P_{k-1} \times P_1$ and go to step E2.

E4. [*Extract SWAP.*] Select a transposition $\tau_{(x,y)} \in T$ such that $\pi_f \cdot \tau_{(x,y)} \in P_{k-1}$.

E5. [*Next SWAP?*] Set $k \leftarrow k-1$ and $\pi_f \leftarrow \pi_f \cdot \tau_{(x,y)}$. If $k = 1$, terminate; otherwise go to step E4. ∎

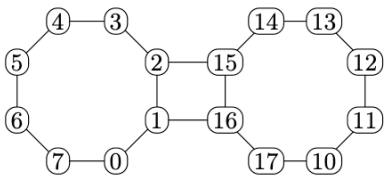
6.7. EXPERIMENTAL RESULTS

We implemented our methods in C++ into the quantum compilation framework *tweedledum*. We evaluate the different approaches with benchmarks on the latest reported hardware architectures based on the superconducting circuit technology. All experiments were run on an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz. As SAT solver, we used a variation of MapleSAT [114].

Benchmarks. As benchmarks, we use a set of 8338 instances of the token swapping problem, which we encountered while mapping quantum circuits from previous works on quantum mapping [157]. These benchmarks contain various functions taken from RevLib [152] as well as quantum algorithms written in Quipper [68] or the Scaffold language [6] (and pre-compiled by the ScaffoldCC compiler [74]). The set of circuits was chosen for evaluation because they are commonly seen in quantum compilation literature and are publicly available for use.

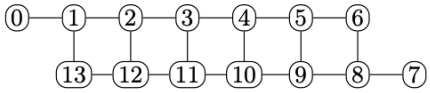
We partition the quantum circuits into parts that can be mapped without SWAPs using an SAT solver as described in [72]. The reconfiguration problems that we studied in this chapter arise from the need to connect these parts into a larger quantum circuit:

We use three real device topologies. The first is Rigetti's 16-qubit quantum computer where the topology consists of two octagons connected by a square:

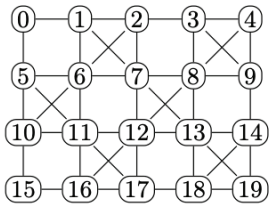


We also use the IBM Q14 Melbourne, a 14-qubit quantum computer with the follow-

ing coupling constraints—we use an undirected graph for this device.



and IBM Q20 Tokyo, a 20-qubit quantum computer:



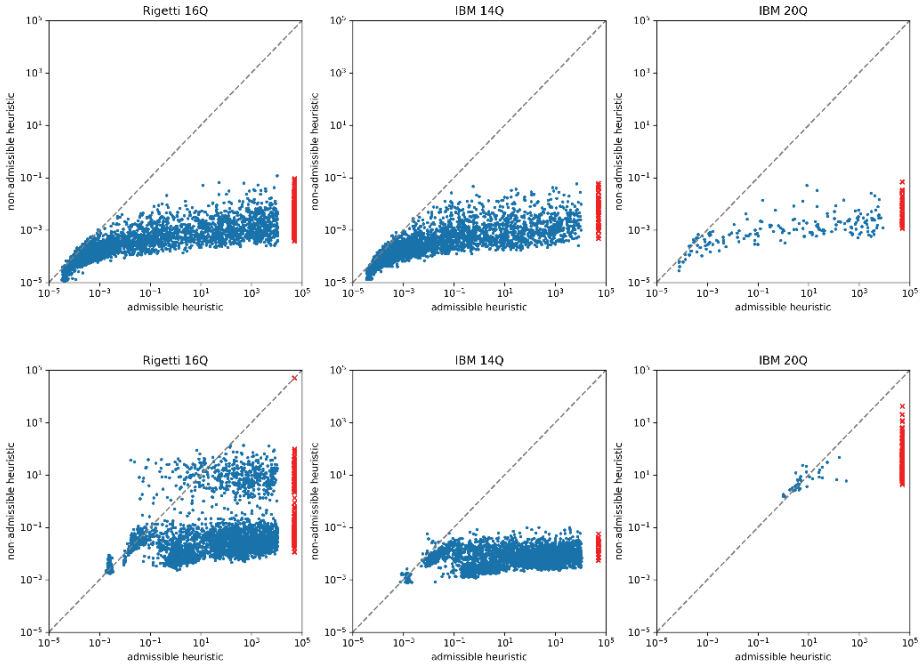


Figure 6.2: Execution time comparison between A*-based algorithm with an admissible heuristic and A*-based algorithm with a non-admissible heuristic. In the top row, the optimization goal is the number of swaps. In the bottom row, the optimization goal is the number of steps.

Method. We try to solve each problem instance for two optimization goals (minimum number of swaps and minimum number of steps) by employing three methods: A*-based with admissible heuristic, A*-based with non-admissible heuristic, and SAT-based. Thus, we run our program a total of 50028 times. Each time we set a timeout of 3 hours.

Figure 6.2 shows the execution time results for solving instances with different optimization goals and using A*-based methods with two heuristics—one admissible and one

non-admissible. In the top row, the goal is to minimize the number of swaps, while on the bottom row, the goal is to reduce the number of steps. As expected, the results show that the non-admissible heuristic has better execution time for both purposes. We note that the non-admissible heuristic is a clear winner when optimizing for the number of swaps; when optimizing for the number of steps, we see that the non-admissable heuristic is still a winner but in many cases performs similarly to the admissible heuristic.

Next, we analyzed the quality of results obtained by employing the A*-based method with a non-admissible heuristic. We calculate the difference in the number of swaps and steps between the obtained result and a known optimal result for each problem instance—which, as we proved, will always be an even number. When solving the token swapping problem, the non-admissible heuristic found an optimal solution in 85.2% and 70% of the benchmarks for Rigetti's 16Q and 20Q, respectively. For IBM's 14Q device, it found solutions using two extra swaps in 87.8% of the cases. The heuristic would use eight additional swaps in the worst cases, but this only happens in less than 1% of the cases. When optimizing for the number of steps, the non-admissible heuristic does not perform well. Indeed, for this case, optimum results are only reached 55.5%, 55.8%, and 13.0% of the time for Rigetti's 16Q, IBM's 14Q, and 20Q, respectively. Further, the number of extra steps might be as high as thirteen.

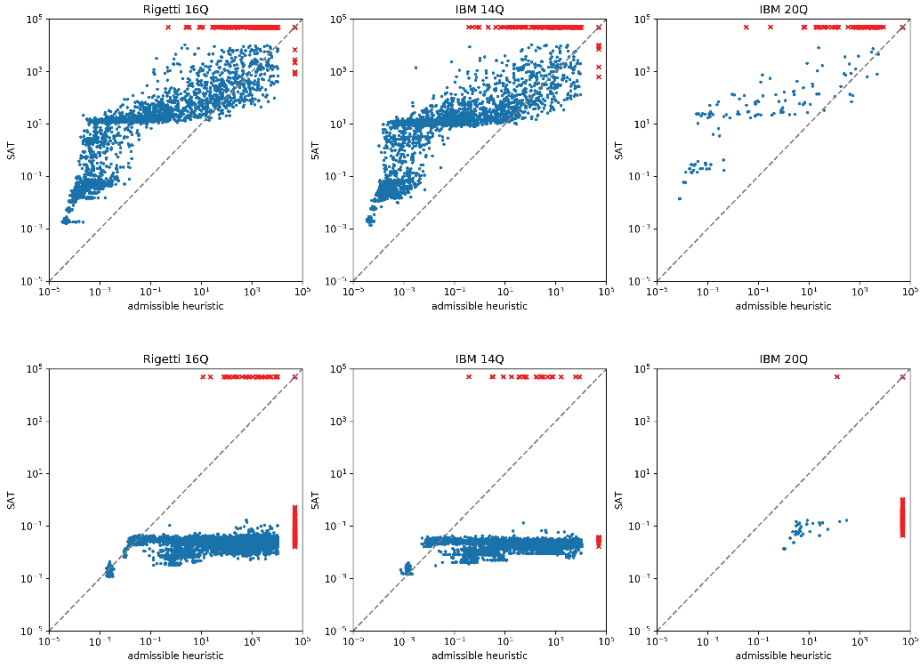


Figure 6.3: Execution time comparison between A*-based algorithm with an admissible heuristic and SAT-based algorithm. In the top row, the optimization goal is the number of swaps. In the bottom row, the optimization goal is the number of steps.

Figure 6.3 shows the execution time results for solving instances with different optimization goals and using the two methods which guarantee optimal results, namely

A*-based with admissible heuristic and SAT-based. The top row shows the results for obtaining an optimal number of swaps, while the bottom row shows results for getting an optimal number of steps. We find that the SAT-based algorithm is better suited for solving the permutation routing via matchings problem, while the A*-based algorithm does better when solving the token swapping problem—although, for many instances, this difference is negligible or non-existent.

6.8. SUMMARY

In summary, we explored different non-admissible heuristics which aggressively prune the search space. We report results for when using $h(s_k) = L + K$, where $K = \sum_{i=1}^{n} d(\Pi(v_i))$ as in Lemma 2.

We presented two symbolic algorithms for solving token swapping and permutation routing via matchings problems. We evaluated the algorithms using a set of practical benchmarks obtained while mapping quantum circuits into different device architectures. The results demonstrate that the A*-based algorithm with an admissible heuristic performs better than the SAT-based method when solving the token swapping problem optimally. The SAT-based method outperforms the A*-based algorithm when optimally solving the permutation routing via matchings problems.

We also evaluated the use of non-admissible heuristics when using the A*-based algorithm. For the token swapping problem, we proved that any result from using a non-admissible heuristic, if not optimal, will deviate from an optimal outcome by at most an even number of swaps. Further, our non-admissible heuristic obtained optimal solutions for most of the benchmarks. When dealing with the permutation routing via matchings problem, using a non-admissible heuristic is not as advantageous.

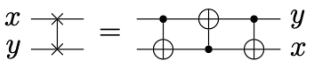
Chapter 7

SAT-based linear synthesis

7.1. MOTIVATION

In the previous chapter, we studied two reconfiguration problems on graphs and showed how both appear in the context of quantum circuit mapping, the task of finding a mapping from virtual instructions to allowed physical ones. The presented techniques synthesize SWAP-based circuits that permute qubits on a coupling graph and thus enable the execution of gates on non-adjacent qubits. Also, these permutation circuits appear prominently in various quantum computing benchmarks [49].

Today's quantum hardware, however, cannot execute SWAP gates directly. Thus, during compilation, we must translate SWAPs into a sequence of CNOT gates using the following identity:



(7.1)

These translated circuits allow us to evaluate the cost in the number of gates and depth more precisely. Also, we note that they can dominate the total costs for many applications. We can improve the implementation of permutation circuits by leveraging the properties of the CNOT gate, as shown in Figure 7.1.

7.2. TECHNICAL BACKGROUND

The family of circuits that permute qubits on a coupling graph is part of the more general family of linear circuits. Let $\mathbb{F}_2$ be the Galois field of two elements. We say that a Boolean function $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ is linear if

$$f(x_1 \oplus x_2) = f(x_1) \oplus f(x_2)$$

for any $x_1, x_2 \in \mathbb{F}_2^n$ where '$\oplus$' is the bitwise XOR. This straightforwardly extends to the n-input and m-output functions $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ where we can define f using a $m \times n$ non-singular Boolean matrix A such that

$$f(x) = A\boldsymbol{x} = \boldsymbol{y}$$

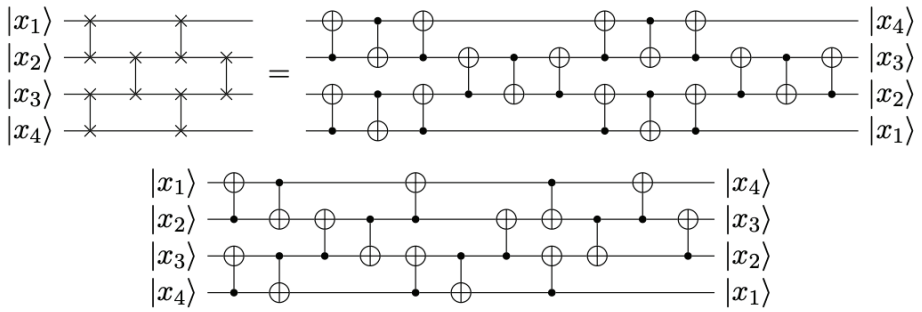


Figure 7.1: This figure shows the results of synthesizing the reversal permutation on a line graph with four vertices while optimizing for the number of operations. On top, we used the method described in the previous chapter to obtain a permutation circuit based on SWAP and then translated it into a CNOT-based one; The result is a circuit with 18 CNOT operations. On the bottom, we directly synthesized a CNOT-based circuit using the technique described in this chapter, which results in a circuit with 15 operations. (We could also have used Kutin's construction for reversal permutations on a line [83].)

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors representing the function's n inputs and m outputs bits respectively. In the case of linear reversible Boolean functions, n equals m, and we have a one-to-one correspondence between the inputs and the outputs. We can derive matrix A from the functions's truth table: the columns correspond to the outputs for each set of inputs containing a single non-zero bit. Note that these matrices are more compact than those used to represent arbitrary gates in quantum computing.

Example 7.1. *The derivation of the linear transformation that reverse a permutation from the reverisble Boolean function that describes it.*

x_4	x_3	x_2	x_1	f_4	f_3	f_2	f_1
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	1	1	0	0	1
1	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1

$$A = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Now, we check with the matrix indeed reverses a permutation:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix}$$

	Description	Primitive	Topologies	Size bounds
[108]	Grouped Gaussian elimination	CNOT	Fully connected	$O\left(\frac{n^2}{\log n}\right)$
[79, 104]	Gaussian elimination with Steiner trees	CNOT	Any	$O(n^2)$
[154]	Recursive elimination of rows and columns	CNOT	Any	$2n^2$
Chapter 6	A* with admissible heuristic	SWAP	Any	Optimal
This	SAT formulation	CNOT	Any	Optimal

Table 7.1: Summary of size-optimizing methods. Bounds are in terms of operation the method is based on (1 SWAP = 3 CNOTs). Note that SWAP-based methods can only synthesize permutations. Also, while out method is optimal, it only practical for small values of n.

7.3. PREVIOUS WORK

While the method described in this chapter can synthesize any linear reversible circuit, we focus the discussion on the synthesis of permutations using CNOTs. Existing methods for synthesizing permutation circuits either optimize for size or depth, using either SWAPs or CNOTs. Some methods are exact but non-scalable, some are tailored to certain topologies, and some are general-purpose heuristics. In Tables 7.1 and 7.2, we summarize prior literature separated by whether they optimize for size or depth.

In [108], Patel et al. designed a block version of the Gaussian elimination algorithm. Instead of treating columns separately, as in straight Gaussian elimination, their algorithm treats blocks of m columns. For each block, their algorithm first tries to eliminate patterns that repeat in different rows. Note that this first step eliminates more than one nonzero element using a single row operation when successful. The second step is to apply Gaussian elimination to handle any remaining nonzero entry below the diagonal. After applying these two steps to all blocks, we end up with a an upper triangular matrix, which we transpose and then the process is repeated to reduce it to the identity. To our knowledge, this algorithm is the best algorithm for synthesizing linear reversible circuits without taking into account connectivity constraints.

More recently, two papers proposed modifying the Gaussian elimination algorithm to synthesize circuits compliant with a connectivity graph given as input of the algorithm [79, 104, 154]. They use Steiner trees to perform a custom Gaussian elimination: the circuit is synthesized column by column, but the process to eliminate nonzero elements is modified to respect the connectivity constraints.

Kutin et al. [83] showed how to synthesize any reversible linear operation while optimizing for depth, but their work is restricted to a path topology. Alon [12] proposed SWAP-based depth optimizing algorithms for a variety of graphs, and Zhang's [156] work improves upon Alon's routing method for trees.

	Description	Primitive	Topologies	Depth bounds
[12]	Graph matchings	SWAP	Various	$3n$
[156]	Caterpillar partition and matchings	SWAP	Tree	$\frac{3}{2}n + O(\log n)$
[83]	Odd-even transposition sort and manual for specific permutations	CNOT	Line	n
[154]	Using n^2 ancillas	CNOT	2D grid	$O(\log n)$
[50]	Divide and conquer	CNOT	Fully connected	$\frac{4}{3}n + 8\log_2(n)$
[21]	Divide and conquer	Reversal	Any	$O(k^2) + \frac{2}{3}r$
Chapter 6	SAT formulation	SWAP	Any	Optimal
This	SAT formulation	CNOT	Any	Optimal

Table 7.2: Summary of depth-optimizing methods. Bounds are in terms of operation the method is based on (1 SWAP = 3 CNOTs). Note that SWAP-based methods can only synthesize permutations. Also, while out method is optimal, it only practical for small values of n.

7.4. CONTRIBUTIONS

The contributions of this chapter are as follows:

- We propose a SAT encoding of the problem and describe a SAT-based algorithm that can find circuits with optimal size or depth using CNOTs.
- We show this algorithm improves over prior optimal SWAP-based methods when synthesizing permutation circuits.
- We use the proposed algorithm to disprove a 15-year-old conjecture by Kutin et al. [83] that reversal is at least as depth-intensive to synthesize with CNOTs as any other permutation on a path.

This work part of a broader paper [37] that will appear at the 59th Design Automation Conference (DAC'2022). An implementation appears in `tweedledum`.

7.5. SYNTHESIS ALGORITHM

We formulate the problem of finding an optimal-depth circuit for a linear matrix representing a reversible function as instances of the Boolean satisfiability problem. In our encoding, we use two kinds of variables. Matrix variables, $m^d_{i,k}$, which indicate whether a matrix entry (i, k) is 0 or 1 at depth d; and CNOT gate variables, $g^d_{c \to t}$, which indicate that a CNOT between qubits c and t took place at depth d.

Example 7.2. *A 3-qubit linear function synthesis would be encoded as such, where a* 3×3 *Boolean matrix represents the linear function over 3 bits, and a CNOT application transforms the matrix by XOR-ing two rows:*

$$\begin{bmatrix} m^d_{0,0} & m^d_{0,1} & m^d_{0,2} \\ m^d_{1,0} & m^d_{1,1} & m^d_{1,2} \\ m^d_{2,0} & m^d_{2,1} & m^d_{2,2} \end{bmatrix} \xrightarrow[g^d_{0\to 2}]{} \begin{bmatrix} m^d_{0,0} & m^d_{0,1} & m^d_{0,2} \\ m^d_{1,0} & m^d_{1,1} & m^d_{1,2} \\ m^d_{0,0} \oplus m^d_{2,0} & m^d_{0,1} \oplus m^d_{2,1} & m^d_{0,2} \oplus m^d_{2,2} \end{bmatrix}$$

Our encoding uses four different types of clauses to constrain the problem such that the solution corresponds to a valid linear reversible circuit:

C1. Each depth that does not hold the target transformation must have at least one CNOT.

$$\forall \{d : \exists (d+1)\}, \sum_{(c,t)\in E} g^d_{c\to t} + g^d_{t\to c} \geq 1.$$

C2. At each depth that has at least one CNOT, each qubit can only be involved in one CNOT.

$$\forall \{d : \exists (d+1)\}, \forall c \in V, \sum_{t\in\delta(c)} g^d_{c\to t} + g^d_{t\to c} = 1.$$

where $\delta(c)$ is the set of qubits adjacent to c.

C3. If at depth d the variable indicating a $CNOT(i,k)$ is true, then all elements of the k-th row at depth $d+1$ must be XOR-ed between the element and its corresponding element in the i-th row at the previous depth d.

$$g^d_{c\to t} \implies \bigwedge_j \left(m^{d+1}_{t,j} = m^d_{t,j} \oplus m^d_{c,j} \right)$$

C4. If at depths d and $d+1$ a matrix entry in the i-th row has different values, then exactly one of the CNOT variables that has i as target must be true.

$$m^{d+1}_{t,j} \neq m^d_{t,j} \implies \sum_{c\in\delta(t)} g^d_{c\to t} = 1$$

The SAT solver only answers whether a given formula is satisfiable or unsatisfiable. Therefore, we need to translate our optimization objective into a series of queries to the SAT solver. In this case, each query "asks" the solver if there exists a circuit that implements the desired transformation using a specified depth. Our implementation incrementally solves the problem: first, we build a formula that encodes a solution with a specified depth using the above constraints; then, if the formula is unsatisfiable, we increment the depth by adding new variables and constraints. We keep incrementing the depth until we find a satisfiable formula to decode and build a linear reversible circuit.

We can use a slightly different encoding to find reversible CNOT circuits with optimal size. In such a case, the only difference lies in the first type of constraints: Instead of

requiring depths to have at least one CNOT, we restrict the number to exactly one.

7.6. DISCUSSION

Our SAT-based solution can be applicable with scalable heuristics to improve their quality [37], but they are also useful on their own. We studied CNOT-depth-optimal solutions for permutations on a path, which was the focus of Kutin et al.'s paper [83]. They conjectured that reversals are at least as hard as any other permutation, and that reversals are synthesizable with depth $2n + 2$. However, by solving for all instances of 8-qubit permutations, we found one permutation that required depth $2n + 3 = 19$, as shown in Figure 7.2, thus disproving the conjecture. In addition, we found that swapping two ends of the path is achievable in depth $n + 7$ for all n, whereas their construction requires depth $n + 8$ for odd n.

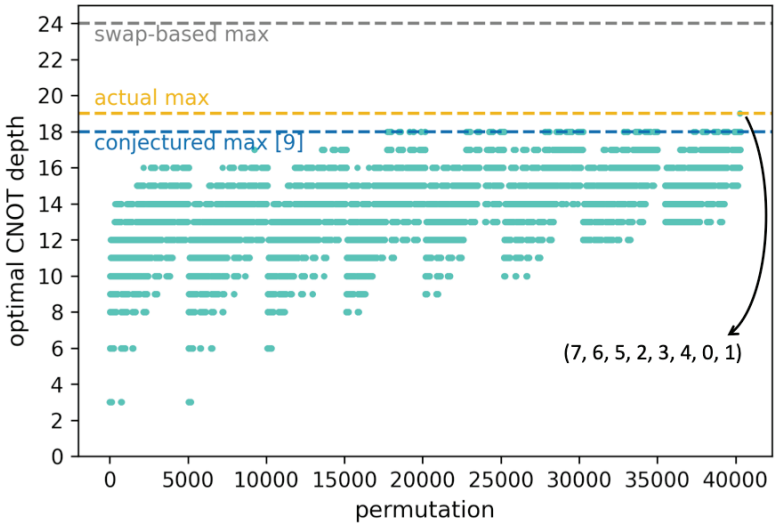


Figure 7.2: Optimal depth of all permutations on an 8 qubit path synthesized by our method.

Part III

Compilation

Chapter 8

Introduction to compilation of quantum programs

The emergence of quantum computers with an increasingly higher number of qubits and longer coherence times marks the beginning of an exciting era in quantum technology, as it empowers us to solve problems that are out of reach for any of the best classical supercomputers. The "Quantum Algorithm Zoo" [76] website holds a comprehensive list of quantum algorithms with their respective speed-up factors. While finding quantum algorithms that yield a significant scaling advantage in time-to-solution, known as a quantum speed-up, over their classical counterparts has been a substantial concern in the quantum computing research community, their concrete implementation and precise resource analysis are still lacking.

Researchers often describe algorithms in a high level of abstraction, using a mixture of natural language, pseudocode, and mathematical formulas. This high-level form facilitates proving correctness and deriving asymptotic complexity estimates while shielding algorithm designers from the low-level complexities and restrictions. However, to execute an algorithm on a quantum computer, a concrete implementation must be provided using low-level abstractions, i.e., the primitive unitary operators supported by the underlying hardware. Note that this is also true in the classical realm.

All computer applications (classical or quantum) rely on software programs built on top of the low-level abstractions provided by a device. One can see a software program as the embodiment of an abstract algorithm in a programming language. In the early days of classical computers, developers had to translate algorithms to programs directly in machine code and, after that, in assembly language. In the early 1950s, Hopper introduced the term compiler to refer to a routine that "compiled" programs from pre-existing pieces [73]; note that it functioned as a loader or linker, not the modern notion of a compiler. Today the term denotes a program that translates a high-level, human-readable programming language into machine code.

While most of today's classical software relies on compilers, they were not widely adopted at first because early ones were somewhat unreliable and generated code that

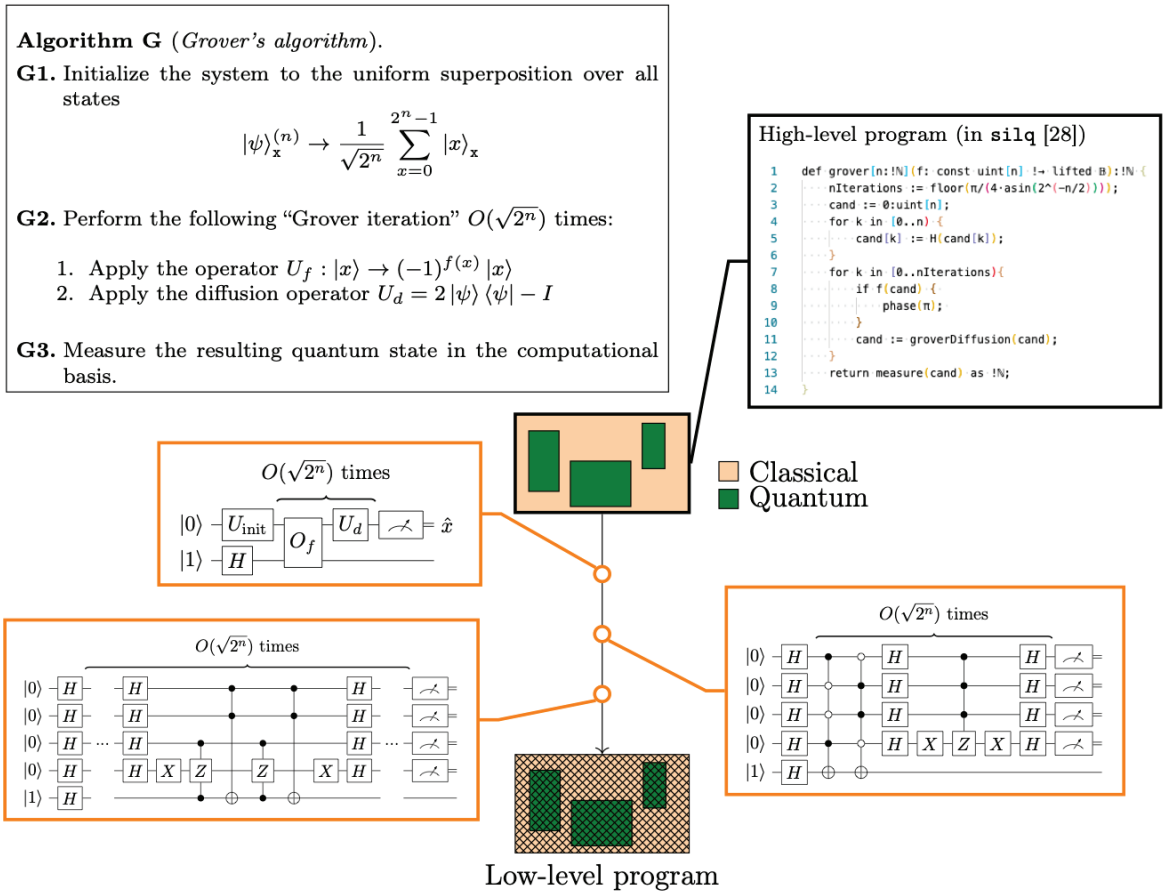


Figure 8.1: Compilation flow overview. A quantum algorithm contains both classical operations and quantum operations. The latter can be given in various forms and sometimes can even be classically defined, e.g., O_f. A compiler translates a high-level program into a sequence of operations executable by a quantum processing unit (QPU).

did not perform as well as what an average developer could write in assembly. Since then, compiler technology has come a long way: today's sophisticated optimizing compilers go to great lengths to ensure efficient, reliable code generation, which is on par, if not better than the code the best human developer could write. Such an improvement has, in turn, enabled classical programming languages to increase significantly in expressive power and sophistication—empowering developers to build more complex programs and more straightforwardly implement algorithms.

In contrast, most programming systems available for quantum computing are intertwined with the quantum circuit model, which means that the developer must still describe an algorithm in terms of primitive unitary operators. Not surprisingly, the implementation of quantum algorithms is very time-consuming, error-prone, and results in non-portable programs, given the diversity of quantum devices.

To explore new algorithms and programs for quantum computing and enhance the developers' productivity, we have seen many quantum frameworks striving to support higher-level abstractions, e.g., `Q#` [145], `Silq` [28], `Quipper` [68], `Qiskit` [150], `Cirq` [55] `PyQuil/Forest` [131], `Pennylane` [26], `ProjectQ` [142], `StrawberryFields` [78]. These frameworks enable the creation of increasingly more complex programs by combining and adapting a small set of known quantum algorithms that use arbitrary technology-independent operations. Nevertheless, given the stringent resource constraints in near-term quantum hardware, the use of higher levels of abstraction is only possible when allied with sophisticated compilation algorithms capable of generating highly optimized low-level circuits. Since circuit compilation and optimization are essential for quantum computing, there are numerous competing compilers and toolkits, e.g., `qiskit-terra` [150], `quilc` [132], `ScaffCC` [74], `staq` [18], and `t|ket⟩` [129]. We refer to [62] and [84] for a surveys of quantum software stacks.

8.1. CONTRIBUTIONS

In the following chapter, will discuss the embodiment of our research contribution: a compiler companion library for the synthesis and compilation of quantum circuits called `tweedledum`. This library adds a unique solution to the fast dynamic field of quantum compilation. In contrast to most solutions, we designed it to enhance other compilers and frameworks, and some of them indeed use it already, e.g., `qiskit-terra` [150], `quilc` [132] and `staq` [18]. The library integrates state-of-the-art algorithms used for quantum compilation, targeting most of the pipeline of a quantum software stack: from the abstract higher algorithmic layers to the physical mapping layer. The following principles guide its design and implementation:

- *Performance.* It must be fast. Compilation performance is critical to productivity. The longer the compiler takes to build a circuit, the more likely developers are taken out of their development flow. On top of that, developers often try to compile a circuit many times while experimenting; hence even short waiting periods can add up to significant disruption.

- *Scalability.* The library must scale up to problem sizes in which quantum circuits outperform classical ones. (Even if such circuits cannot run on today's quantum devices, accurate resource estimation is necessary to help guide the development of new hardware.)

- *Customizability.* The intermediate representation (IR) is based on a handful of fundamental concepts, leaving most of it entirely customizable. For example, `tweedledum` does not have a fixed set of operators nor imposes any restriction on how their effects are defined.

We also present less significant contributions to quantum circuit mapping, adaptative decomposition of quantum gates, and ways of composing different algorithms to create effective compilations flows.

Chapter 9

The `tweedledum` library

The library aims to be easy to use, comprehend, and extend. Therefore, we base its intermediate representation only on a handful of fundamental concepts: wires (qubit and bit), operator, instruction, and circuit. It is worth noting that the operator concept is entirely customizable. This customizability ensures the library can adapt to a fast-changing research environment yielded by industry and academia exploring several technologies that support quantum computation.

We implement `tweedledum` [119] as an open-source library in C++-17 and provide Python bindings for easy integration into existing compilers/frameworks. The remainder of this work describes `tweedledum`'s core data structures and briefly introduces most of its state-of-the-art techniques for synthesizing and compiling quantum circuits. To demonstrate the many features implemented in `tweedledum` and its design space exploration capabilities, we present two non-trivial use cases. The ultimate goal of the `tweedledum` is to create an open-source environment in which state-of-the-art quantum compilation techniques can be developed and compared.

9.1. THE INTERMEDIATE REPRESENTATION

In `tweedledum`, the standard intermediate representation is a quantum circuit. A circuit has a set of wires (qubits and bits) and a sequence of operators applied to those wires, the so-called instructions. An operator is an abstract effect that may modify the state of a subset of wires. An instruction is an operator applied to a specific subset of qubits and bits. In other words, to represent a quantum computation, we first create an empty circuit to which we add qubits and bits. Then we create instructions by applying operators to these qubits and bits.

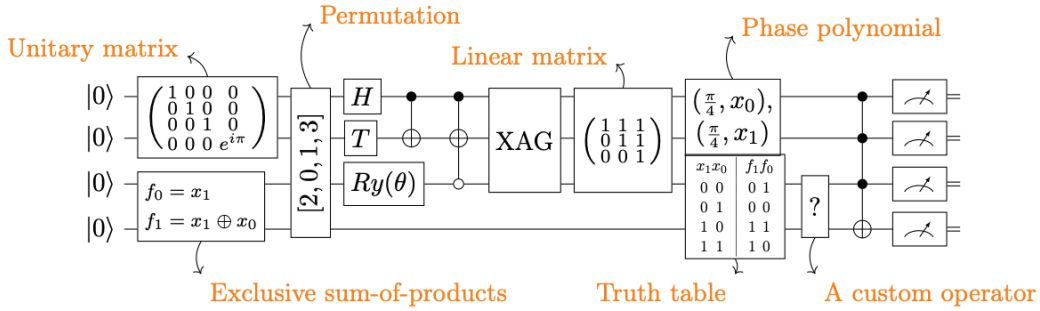
9.1.1 Fundamental concepts

Wires (qubits and bits). We represent qubits and bits using memory semantics. Meaning that instructions act on references to qubits (and bits) and do not consume their value, i.e., state; we say that they affect their state via side effects. A benefit of

using memory semantics is that the intermediate representation inherently prevents a program from violating the no-cloning theorem [153].

Operators. Conceptually, an operator is an effect that we can apply to a subset of qubits and bits, which, most often, this effect is unitary evolution. To ensure a high level of customizability, the library does not have a fixed set of operators nor imposes restrictions on how their effects are defined. On the contrary, it encourages and facilitates the implementation of user-defined ones. Indeed, one of the main strengths of `tweedledum` lies in its use of a uniform concept, known as *Operator*, to enable describing different levels of abstractions and computations.

Example 9.1. *We can define high-level operators such as the truth table operator, the permutation operator, or low-level operators like the Hadamard operator. The following circuit illustrate how flexible is the intermediate representation.*



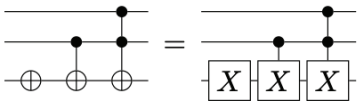
The library imposes two minimal requirements for the Operator concept. Namely, every Operator must be identifiable, even as an "unknown," and operators must target at least one qubit. (Note that the number of targets is intrinsic to an operator; the number of controls is not—we explain the reason for this when introducing instructions.)

There is also a special set of compiler-internal operators has a null effect on wires. We call them meta-operators. One example is the `Barrier` operator: it acts as a compiler directive to separate pieces of a circuit during compilation. Any optimizations or rewrites are constrained to only act between them. The execution of a quantum circuit should completely ignore all meta-operators.

Instructions. An instruction is the embodiment of the application of an operator to a specific subset of wires. The number of qubits must be equal to (or greater than) the number of targets required by the operator. If the latter, the extra qubits are considered controls. This design choice aims to counter-balance the easiness of defining new operators, which might lead to an explosion in their number.

Example 9.2. *In `tweedledum`, the instructions* NOT, CNOT *and* TOFFOLI *are the*

same operator X *applied to a different number of qubits.*



Circuit. In `tweedledum`, a quantum circuit is a directed acyclic graph with labeled vertices and labeled edges. Vertices correspond to instructions. Their label determines which operator the instruction applies. The edges encode input/output relationships between instructions, and their label indicates the qubit (bit) associated with this relationship. Its underlying implementation also allows it to be treated as a simple list of instructions (netlist).

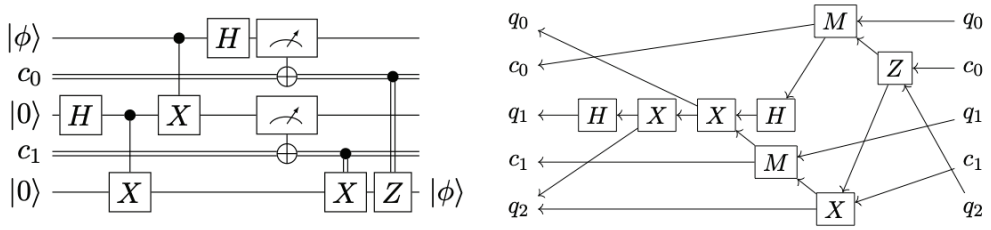


Figure 9.1: Quantum circuit representation for teleportation of a quantum state.

9.2. SYNTHESIS

In this section, we briefly describe the various synthesis methods implemented in the library:

- `a_star_swap_synth` [Chapter 6]
- `cx_dihedral_synth` [95]
- `decomp_synth` [52]
- `diagonal_synth` [125]
- `gray_synth` [17]
- `lhrs_synth` [138]
- `linear_synth` [108]
- `pkrm_synth` [Chapter 4]
- `pprm_synth` [Chapter 4]
- `sat_linear_synth` [Chapter 7]
- `sat_swap_synth` [Chapter 6]
- `spectrum_synth` [135]
- `steiner_gauss_synth` [79, 18]
- `transform_synth` [97]
- `xag_synth` [Chapter 5]

An exciting feature of `tweedledum` is the ability to insert classical logic directly into quantum circuits and to synthesize it during compilation. Moreover, the library provides means for a user to define such classical logic as a Python function. By combining most of the EPFL logic synthesis libraries [136], we can create compilation flows that handle the translation of this Python function into a sequence of elementary quantum operators.

If not given as a Python function, `tweedledum` also accepts Boolean functions provided in others forms. Both `decomp_synth` and `transform_synth` take as input a reversible function in the form of a permutation. (Note that a reversible function implements a bijective mapping between input and output binary signals; thus, a reversible function is a permutation of its inputs.) The decomposition-based synthesis technique iteratively decomposes the function into simpler functions based on the Young subgroup decomposition [52]. Transformation-based synthesis keeps changing the function by applying multiple controlled X operators until it becomes the identity function.

More often, however, we will be dealing with irreversible Boolean functions. Given an irreversible function f, it is known that there must exist a reversible Boolean function $f_r : \{0,1\}^{n+1} \mapsto \{0,1\}^{n+1}$ such that

$$f_r(x, y) = (x, y \oplus f(x)),$$

where $x = x_0, \ldots, x_{n-1}$ and '$\oplus$' refers to the XOR operation. (For the sake of clarity, we limit the discussion to single-output Boolean functions, but the technique can be extended to accommodate multiple-output functions.) Such an embedding is also referred to as Bennett embedding [25], and implies the existence of the following quantum operation:

$$U_f : |x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle$$

The operation U_f is also known as a single-target operator. Single-target operators describe complex operations that cannot generally be implemented natively on a quantum computer. `tweedledum` provides three techniques to synthesize single-target operators directly. Starting from a functional representation of f, i.e., a truth table, both `pkrm_synth` and `pprm_synth` techniques synthesize a particular case of an exclusive sum-of-products (ESOP) expression for f. We can easily translate such ESOP into a cascade of multiple-control X operators. These techniques, however, are only applicable to small Boolean functions as they can be both very time-consuming and generate a quantum circuit with a prohibitive number of instructions [123, 113]. `spectrum_synth` [135] uses the Rademacher-Walsh spectrum of a truth table to generate a circuit over the operator set Clifford+Rz directly.

For a more scalable solution, we combine these direct methods with the hierarchical synthesis approaches. The latter allows us to achieve scalability by decomposing the initial function into small parts suitable for functional synthesis. The synthesis method generates a reversible circuit for each part and combines them following the structural representation of the function. The combination of subcircuits might require additional qubits, which store intermediate computation steps. Given an irreversible Boolean function f they find an $(n + 1 + a)$-qubit quantum circuit that realizes the unitary

$$U_f : |x\rangle |y\rangle |0\rangle^a \mapsto |x\rangle |y \oplus f(x)\rangle |0\rangle^a$$

where $a \geq 0$, which means that the synthesis algorithm can use the a additional qubits to

store intermediate computations. `lhrs_synth` [138] and `xag_synth` [120] are examples of hierarchical synthesis.

Linear reversible circuits form a subclass of quantum circuits in which implementation requires only CNOT operators. Synthesis methods aiming to reduce the size of these circuits play an essential role in `tweedledum` since other algorithms depend on them. Given a binary matrix describing the classical function, the library has two techniques for synthesizing such circuits. Both methods rely on a modified implementation of Gaussian elimination, which yields asymptotically optimal circuits. They differ in that the `steiner_gauss_synth` [79, 18] algorithm can synthesize circuits that respect the connectivity constraints of device architectures. We also provide two techniques, `a_star_swap_synth` and `sat_swap_synth` [124], to synthesize an even more constrained class of linear reversible circuits: those composed entirely of SWAP operators.

The sum-over-path form can help a compilation flow to circumvent possible structural biases present on a quantum circuit. As we have seen, it is straightforward to obtain it from a quantum circuit over CNOT and $P(\theta)$ operators. `tweedledum` implements two synthesis algorithms that take as input the sum-over-path form. Namely, `gray_synth` [17] and `cx_dihedral_synth` [95]. The latter is a Boolean satisfiability-based method that guarantees optimality in the number of operators but is not scalable. As we will discuss later, a decomposition technique is better suited when the sum-over-path form is not sparse, i.e., when most or all linear combinations are present.

Finally, we note that the library provides one synthesis method to handle constrained unitary matrices. `diagonal_synth` [125] implements a algorithm to synthesize circuits that perform arbitrary controlled phase-shift operations, which must be given in the form of a diagonal unitary matrix.

9.3. COMPILATION

9.3.1 Utility

Passes in this category provide simple utilities that do not fit any other category, which more complex passes might either require or significantly benefit from using it. For example, since `tweedledum` does not modify circuits in place, all passes that modify a circuit must do a shallow duplication, i.e., create a new circuit with the same wires. There are also passes to reverse and invert circuits. The `reverse` pass creates a new circuit with the instruction applied in the reverse topological order. Inversion is similar, but with the addition of applying the adjoint instruction.

We also have an instruction canonicalization pass. The goal of canonicalization is to make optimizations more effective. Very often, we can write instructions in multiple forms. For example, we can write them with equivalent operators $T = P(\frac{7\pi}{8}) = P(-\frac{\pi}{4})$; or apply an operator to a permutation of the same wires $\mathrm{SWAP}(q_0, q_1) = \mathrm{SWAP}(q_1, q_0)$. The latter case requires caution as the equivalence depends on the specific operator: While the permutation of controls always yields equivalent instructions, this is not always the case for targets' permutation. Canonicalization means selecting one of these forms to

be canonical and then going through a circuit and rewriting all instructions equivalent to the canonical form into the canonical form. Thus, canonicalization allows optimization passes that look for specific patterns to focus only on the canonical forms rather than all forms.

9.3.2 Decomposition

During the compilation of a technology-independent quantum circuit, high-level instructions often need to be broken down into a series of lower-level instructions. We already talked extensively about synthesis: a process that can lower abstraction by inputting a high-level quantum functionality specification and outputting a quantum circuit. Here, we introduce decomposition, another procedure with similar capability.

We define decomposition as systematically breaking down high-level instructions into a series of lower-level ones. We emphasize "systematically" because it is the characteristic that differentiates decomposition from synthesis. A decomposition technique builds a lower-level implementation by applying some construction rule(s). Compared to synthesis, decomposition techniques are faster and produce predictable results, i.e., we know in advance the resulting number of instructions and qubits. The following table lists all decomposition algorithms currently provided in `tweedledum`, along with a short description.

	Short description
`barenco_decomp`	This decomposition pass is capable of breaking down Pauli-(X/Y/Z) instructions controlled by three or more qubits. We based our implementation on the principles presented by Barenco et al. in [22].
`bridge_decomp`	Trivially decomposes a bridge instruction into a series of CNOT instructions.
`euler_decomp`	Decompose a arbitrary one-qubit instruction, given as unitary matrix, as a sequence of at most three R_z and R_y operators. This is due to the ZYZ decomposition: Given any 2×2 unitary matrix U, there exist angles γ, ϕ, θ, and λ satisfying: $U = e^{i\gamma} R_z(\phi) R_y(\theta) R_z(\lambda)$ [22].
`linear_decomp`	Decompose an instruction given as a sum-over-path form. This method works best when the form is not sparse, i.e., when most or all linear combinations are present.
`parity_decomp`	Trivially decomposes a parity instruction into a series of CNOT instructions.

Table 9.1: Decomposition algorithms.

The cost to decompose a multiple controlled Pauli instruction depends on whether

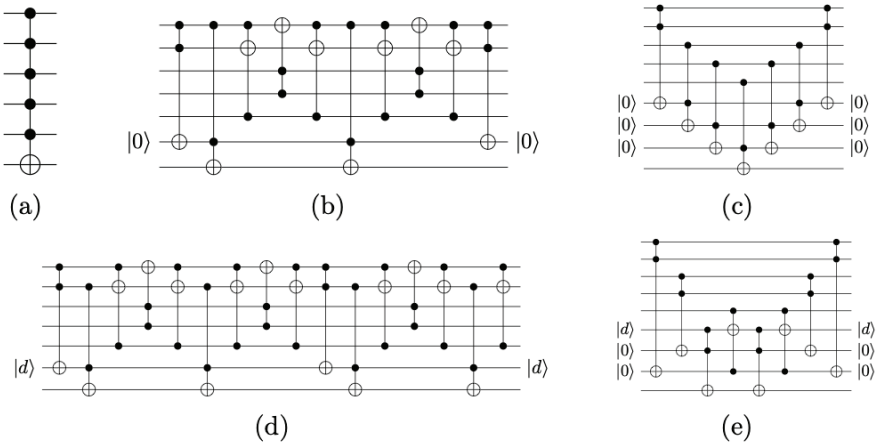


Figure 9.2: Different ways of decomposing a multiple controlled X instruction (a), depending on the number of ancillae available.

the circuit has clean ancillae available. (Here, we measure cost by the number of instructions.) For example, the circuit might have enough clean ancillae to allow a "v-chain" decomposition, as illustrated in Figure 9.2. If that is not the case, we need to use the more costly "dirty-ancilla" decomposition (based on Lemmas 7.1 and 7.2 from [22]). An interesting feature of our implementation of `barenco_decomp` is its adaptability. It uses a "v-chain" for as long as there is a clean ancilla, then switches to "dirty-ancilla." Furthermore, depending on the user configuration, the algorithm can automatically add clean ancillae and use relative phase instructions [92] for intermediate computations.

9.3.3 Mapping

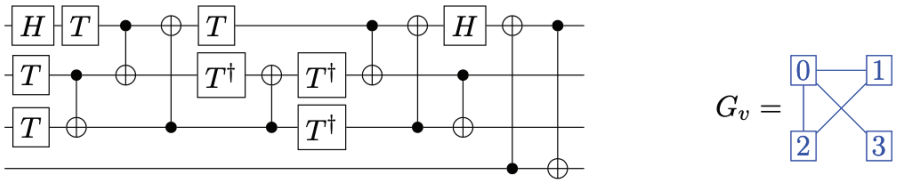
The physical qubits in most quantum hardware are not fully connected, which means that not every pair can participate in the same physical instruction. These connectivity restrictions are known as coupling constraints. In `tweedledum`, the `Circuit` class has no mechanism to enforce them, i.e., we can apply an operation between any pair of qubits. Therefore, synthesized or constructed circuits typically cannot be directly executed on a quantum device. We refer to them as unmapped circuits and define virtual (or logical) qubits and instructions as those present in them. The task of finding a mapping of virtual instructions to allowed physical instructions is known as quantum circuit mapping. The completion of this task is not always possible without applying additional operators to the circuit.

Formally, we model the connectivity requirements of an unmapped circuit using an undirected graph $G_v = (V, E_v)$ where V is the set of virtual qubits $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and $E_v \subseteq \binom{V}{2}$ is a set of qubit pairs $\{v_i, v_j\}$ used by the instructions in the circuit. (Note that one-qubit instructions can be safely ignored since the coupling constraints do not affect their mapping.) We model the coupling constraints in a similar way. A unidirected graph (P, E_p) where $P = \{p_0, p_1, \ldots, p_{m-1}\}$ is a set of physical qubits and an edge $\{p_u, p_w\} \in E_p \subseteq \binom{P}{2}$ means that an instruction can be executed using the two

physical qubits p_u and p_w. The library divides mapping into two sub tasks: placement and routing.

Placement. The goal is to find a subgraph monomorphism $\pi : V \mapsto P$ that respects $(v_i, v_j) \in E_v \implies (\pi(v_i), \pi(v_k)) \in E_p$. If such a morphism exists, then we call π a perfect placement and the mapping task requires only a relabeling of the virtual qubits to physical qubits, i.e., replace each virtual qubit v_i in the unmapped circuit by a physical qubit $\pi(v_i)$. Otherwise, we must resort to routing. Table 9.2 shows the placement algorithms implemented `tweedledum`.

Example 9.3. *Suppose we are given the following circuit and its corresponding connectivity requirements:*



When we try to map this circuit on the IBMQ5 Yorktown device, we can find a perfect placement:

However, we cannot find a perfect placement for IBMQ Belem device:

Both `ApprxSatPlacer` and `SatPlacer` try to find a perfect placement by solving a Boolean satisfiability problem. We encode a qubit placement problem as a Boolean function. This function is satisfiable if and only if there is a perfect qubit placement. An SAT solver determines the Boolean function's satisfiability. When the problem is satisfiable, the solver provides a satisfying assignment from which we extract the perfect placement.

It is essential to properly consider how to encode the placement problem as a Boolean function because the encoding can have an enormous effect on the speed with which an

	Short description
`ApprxSatPlacer`	At first, it tries to find a perfect placement for the qubits using Boolean satisfiability. On failure, it relax the problem's constraints and tries to approximate a perfect one.
`LinePlacer` [48]	Transforms a connectivity requirements graph G_v into a graph in which each component is a line. Then tries to map this graph to the coupling graph as one long line starting from a high degree physical qubit and greedily choosing the highest degree available neighbor.
`RandomPlacer`	Randomly place virtual qubits into physical qubits.
`SatPlacer`	Try to find a perfect placement using Boolean satisfiability.
`TrivialPlacer`	Place each virtual qubit v_i into physical qubit p_i.

Table 9.2: Placement algorithms.

SAT solver can find an answer to our problem. Both `ApprxSatPlacer` and `SatPlacer` use two kinds of clauses to constrain the problem such that a solution corresponds to one correct placement—namely, the qubit constraints and the instruction constraints. The qubit constraints guarantee that (1) each virtual qubit is placed on precisely one physical qubit and (2) at most one virtual qubit is placed on a physical qubit. The instruction constraints ensure that virtual qubit pairs used by the instructions in the circuit are placed on adjacent physical qubits.

The encoding used by `SatPlacer` has only one kind of variable, which expresses whether a virtual qubit is placed on a physical qubit. Let x_{vp} be a Boolean variable which indicates whether a virtual qubit v is placed on a physical qubit p. We can express the qubit constraints as

$$(1)\ \sum_{p \in P} x_{vp} = 1,\ \forall v \in V, \quad \text{and} \quad (2)\ \sum_{v \in V} x_{vp} \leq 1,\ \forall p \in P.$$

Given a physical qubit p_m, let $\delta(p_m)$ is the set of physical qubits adjacent to p_m. Then we can formally express the instruction constraints as

$$x_{v_i p_m} \implies \sum_{p_n \in \delta(p_m)} x_{v_j p_n} = 1,\ \forall p_m \in P,\ \forall \{v_i, v_j\} \in E_v.$$

As seen in the results in Section 9.5, `SatPlacer` can quickly find a perfect placement for each of the 900 benchmarks in QUEKO [146]. While such results are impressive, the technique is limited to either returning a perfect placement or no placement. Therefore, if `SatPlacer` fails to find a perfect placement, then another placement technique must be employed. `ApprxSatPlacer` addresses this limitation by adding another kind of variable that slightly changes the encoding. These new variables allow the activation (or deacti-

vation) of the instruction constraints. Let a_{v_i,v_j} be a Boolean variable which indicates whether to activate or not a instruction constraint. We rewrite `SatPlacer`'s instruction constraints as

$$a_{v_i,v_j} \wedge x_{v_i p_m} \implies \sum_{p_n \in \delta(p_m)} x_{v_j p_n} = 1,\ \forall p_m \in P,\ \forall \{v_i, v_j\} \in E_v.$$

At first, technique `ApprxSatPlacer` tries to find a perfect placement for the qubits, i.e., it assumes all instruction constraints are active. Suppose the SAT solver concludes that it is impossible to find such a placement. In that case, it returns a conflicting clause that can help to identify which instruction constraints cannot be satisfied together. We must relax the constraints by deactivating at least one of them. We implemented two heuristics to choose which instruction constraints to deactivate:

- We create instruction constraints while iterating over the instructions of a circuit, which intrinsically creates an ordering. Our first heuristic chooses first to deactivate constraints that appeared later.
- While iterating over the circuit, we also keep track of how many instructions depend on a given constraint. That is the number of instructions that depend on virtual qubits v_i and v_j to be adjacently placed. We call this metric the constraint's weight. The second heuristic chooses first to deactivate constraints with a lower weight.

We repeat the process of calling the SAT solver while deactivating constraints until we can find a placement that satisfies the active constraints. Note that such a placement might be partial, i.e., some virtual qubits are left unplaced.

Routing. Given an initial placement, a router transforms a circuit so that all two-qubit instructions operate on physically adjacent qubits. Thus, our goal becomes finding a way of mapping a given circuit on a given device architecture with low overhead, whether in the number of additional instructions or depth of the resulting circuit.

All routers implemented in `tweedledum` guarantee the compilation of any quantum circuit to any architecture represented as a simple connected graph. They are, therefore, completely hardware agnostic. The rest of this subsection focuses on our implementation of `LazyRouter`, a modified implementation of `SabreRouter` that can work with partial initial placements. For the routing algorithm to proceed, we require an initial placement of virtual qubits into physical qubits. The routing algorithm iteratively constructs a new circuit that conforms to the desired architecture constraints by visiting all instructions of the original circuit.

The algorithm starts by visiting all instructions that have not an unvisited predecessor in the circuit. It immediately adds to the new circuit all instructions compatible with the current placement—i.e., the one-qubit instructions acting on placed qubits and two-qubit instructions action on adjacently placed qubits. If an instruction requires unplaced qubit(s), the routing algorithm either postpone its addition to the new circuit or place

	Short description
`BridgeRouter`	This router maps the circuit by trivially replacing all CNOT instructions operating on physically separated qubits by Bridge instructions.
`LazyRouter`	A modified implementation of `SabreRouter` which can work with partial initial placements. Unplaced qubits are lazily placed, i.e., we placed them only when they participate in a two-qubit instruction or at the end of the routing process.
`SabreRouter`	This router implements the SWAP-based routing heuristic described in [86] (Algorithm 1). The heuristic aims to minimize the number of SWAPs inserted and the depth of the circuit. It requires a complete initial placement.

Table 9.3: Routing algorithms.

the qubit(s). It chooses to postpone one-qubit instructions and to deal with two-qubit instructions the following way:

- If the instruction acts on one unplaced qubit, we place this qubit in its partner's nearest available physical qubit.
- If the instruction acts on two unplaced qubits, we try to place both in the closest pair of available physical qubits.

The routing algorithm checks for postponed instructions whenever it places a qubit. Since postponed instructions are guaranteed to be one-qubit, they are immediately added to the new circuit. The algorithm adds SWAPs to the new circuit whenever it encounters two-qubit instructions incompatible with the current placement, i.e., the instruction acts on non-adjacently placed qubits. The selection of SWAPs follows the same heuristic as `SabreRoute`, detailed in [86]. The routing process ends when all original instructions are present in the new circuit.

9.3.4 Optimization

With the limited space and time resources available on current quantum devices, aggressive circuit optimization techniques are essential to extract all performance out of the machines. They often play a crucial role in whether a circuit can or cannot execute in a device or a simulator. Furthermore, they provide more accurate resource estimates, which might guide quantum algorithms and hardware development.

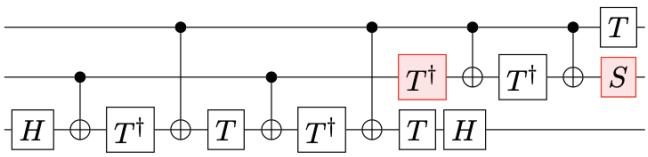
`tweedledum` provides circuit optimizations as a set of orthogonal compilation passes, which can be composed into compilation flows. Optimization techniques range from purely structural, relying only on the relationship between instructions on a quantum circuit representation, to purely functional, e.g., when they rely on resynthesizing a circuit

from a unitary matrix. There exists a significant trade-off between the quality of results and scalability. On the one hand, structural transformations offer better scalability at the cost of inferior quality of results. On the other hand, functional optimizations offer the contrary: better quality of results and poor scalability.

Instruction cancellation. As the name implies, this pass performs basic instruction cancellation: it traverses a circuit and removes pairs of adjacent adjoint instructions. This optimization is purely structural, and its effectiveness is highly dependent on instruction canonicalization. For example, a pair of adjacent instructions which apply the operators T and $P(-\frac{\pi}{4})$, respectively, will not be optimized.

Phase folding. This optimization pass extends [19]'s implementation of T-par's T-count optimization algorithm to enable merging parameterized phase operators and handling arbitrary operators. The implementation handles operators not belonging to set $\{X, \text{CNOT}, \text{SWAP}, P(\theta)\}$ conservatively. It ignores their phase contribution, meaning that this pass only keeps track of phase polynomial terms that are trivially mergeable.

Example 9.4. *Take the following circuit implementation of the Toffoli gate found in [105]:*



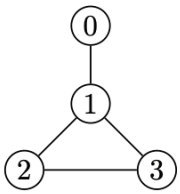
Phase folding can optimize this circuit by merging the highlighted gates into a single $\boxed{T}$*—which can be place on either original gates' locations.*

Linear resynthesis. The instruction cancellation pass suffers from structural bias: the input/output relationship between instructions (the structure) strongly influences the quality of results. Resynthesis techniques circumvent this problem by traversing the circuit, searching subcircuits that they know how to represent functionally and synthesize. In the linear resynthesis pass, we first identify linear subcircuits and represent their functionality as a binary matrix. Then we try to find a less costly implementation by resynthesizing a new subcircuit using `linear_synthesis` or our `sat_linear_synthesis` algorithm, described in Chapter 7. If the new subcircuit has fewer instructions than the original, we use it in the circuit. Otherwise, we ignore it. Users can also replace zero gain subcircuits since they might help other optimizations escape their structural bias. Furthermore, we can apply this pass to a mapped circuit using either the `steiner_gauss_synth` or our's algorithm because both synthesize circuits while respecting coupling constraints.

9.4. SHOW CASE: BOOLEAN FUNCTION COMPILATION

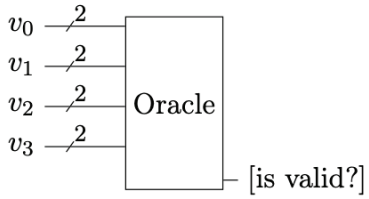
This section will use an example of use Grover's algorithm to solve an instance of the vertex coloring problem to illustrate how to build compilation flows using `tweedledum`.

Let's start by explaining the problem we want to solve. Vertex coloring is the problem of assigning colors to vertices of a graph such that adjacent vertices are not of the same color. Let $G = (V, E)$ be an undirected graph, where V is the graph's vertex set, and E is the set of edges. In our illustrative example, we want to color the following graph:



While coloring the above graph is easy, we know this problem's decision version, where we are interested in whether a graph can be colored with k colors, is NP-complete [77]. Its optimization version, where we look for the smallest number of colors required to color a graph, is NP-hard. In our example, we want to know whether it is possible to color the graph using three colors, i.e., $k = 3$.

We can solve the problem by assuming the existence of an oracle capable of identifying correct solutions. This oracle takes $|V|$ binary strings as input, each representing a color assigned to a vertex, and returns a Boolean value indicating whether the assignment is valid, i.e., it solves the problem:



Now, we can query this oracle with all input combinations until we find one for which the output indicates that the input is a valid coloring of the vertices. Classically, we will query the oracle $O(2^n)$ times in the worst case, where n is the total number of input bits—in our example $n = 8$. However, a quantum computer can solve such a problem with high probability by querying the oracle only $O(\sqrt{2^n})$ times.

Boolean modeling. Before delving into how a quantum computer can achieve such a feat, let's look at how to model our oracle as a Boolean function. We assign to each vertex a binary string that represents a color and formulate the following constraints:

- Every vertex must have a valid color assigned to it.
- Two adjacent vertices cannot have the same color.

We define the three colors as A ('01'), B ('10'), and C ('11')—note that '00' is an invalid color. For each vertex $i \in [3]$, we create a variable, which is a bit string of length 2. The oracle can then be modeled as a Boolean function $f(v_0, v_1, v_2, v_3)$ that evaluates to 1 (true) only for those variable assignments representing a graph coloring satisfying all constraints.

Grover's algorithm [69]. The basic idea of Grover's algorithm is to invert the phase of the desired basis state, and then invert all the basis states about the average amplitude of all the states. The algorithm uses $n+1$ qubits, where the first n of them are initialized with $|0\rangle$ and the last one is initialized with $|1\rangle$. The initialization operator U_{init} creates a uniform superposition of all classical states that are inputs to the oracle function f and then repeatedly applies two operators to the state:

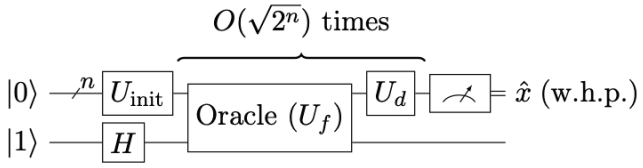
1. U_f, a bit oracle of the Boolean function f that is cast into a phase oracle.

2. The second operator

$$U_d = U_{\text{init}} \cdot (2\,|0^n\rangle\,\langle 0^n| - I_{2^n}) \cdot U_{\text{init}}^\dagger$$

   is a $2^n \times 2^n$ diffusion operator. Here, $|0^n\rangle$ is the classical state represented by the bit string with n zeros, I_{2^n} is the identity operator of size $2^n \times 2^n$, and $U_{\text{init}}^\dagger$ is the adjoint operator of U_{init}.

In circuit form:



(9.1)

Note that Grover's original paper worked mostly with oracles that evaluated to true for only one input assignment, i.e., when only one solution exists. Grover briefly considered the possibility of multiple solutions but provided no details concerning the efficiency of his method. A later work by other researchers offered such details [32].

Implementation. The circuit in (9.1) represents an abstract implementation of Grover's algorithm that we need to make concrete to solve our problem. We start by defining the oracle's behavior using a Boolean function, which we define using Python:

```
def f(v0, v1, v2, v3 : BitVec(2)) -> BitVec(1):
    not_00 = (v0 != '00') and (v1 != '00') and
             (v2 != '00') and (v3 != '00')
    c_01 = (v0 != v1)
    c_123 = (v1 != v2) and (v1 != v3) and (v2 != v3)
    return not_00 and c_01 and c_123
```

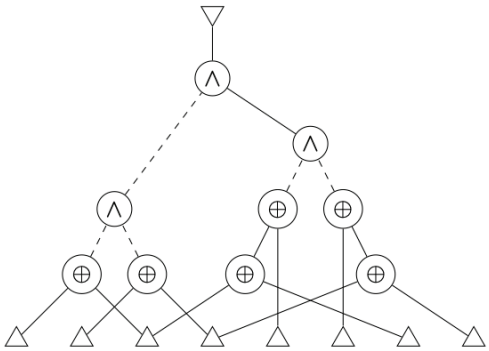
This function returns '1' (true) only when all vertices have a valid color (i.e, either '01', '10' or '11'), and no adjacent vertices have the same color.

In this sort of problem, there is often a degree of commonality between various non-solutions. For example, one typically knows beforehand that some assignments (or combinations of assignments) of the variables are inconsistent, i.e., violate one or more of the constraints, and cannot participate in any solution. In our example, we know that '00' is not a valid color. Thus, we can simplify the implementation of our function by ensuring we will never call it with a input combination in which at least one of the input variables is '00':

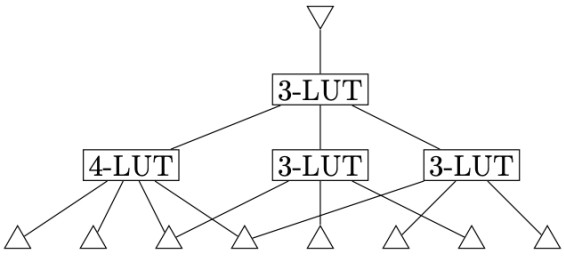
```
def f(v0, v1, v2, v3 : BitVec(2)) -> BitVec(1):
    c_01 = (v0 != v1)
    c_123 = (v1 ^ v2 ^ v3 == '00')
    return c_01 and c_123
```

Note that the naïvely implementation of U_{init} creates the superposition of all classical states by applying the Hadamard gate to all qubits. If we want to use this optimized function, however, we to guarantee that our initialization only creates the superposition of all *valid* computational basis states, that is, all states for each '00' is not assigned to a vertex. We will not further discuss U_{init}'s implementation and assume it to be given.

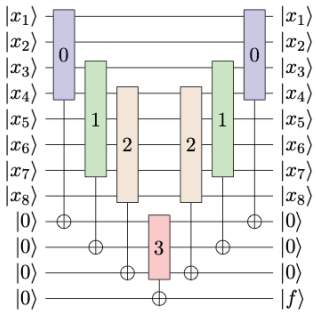
In the following, we describe two flows to compile this high-level definition of f into a quantum circuit. Both flows start in the same way: They take the Python function and transform it into a XAG representation:



LUT-based flow. This flow transforms the XAG into a k-feasible Boolean logic network (k-LUT networks). In simpler words, it takes the 8-input function and represents it in terms of smaller Boolean functions that have at most k inputs, in this case we chose $k = 4$ and obtain the following network:



Then the compilation proceeds in two steps: first, each k-LUT is translated to a single-target gate with at most k control lines and combined in a reversible logic network:

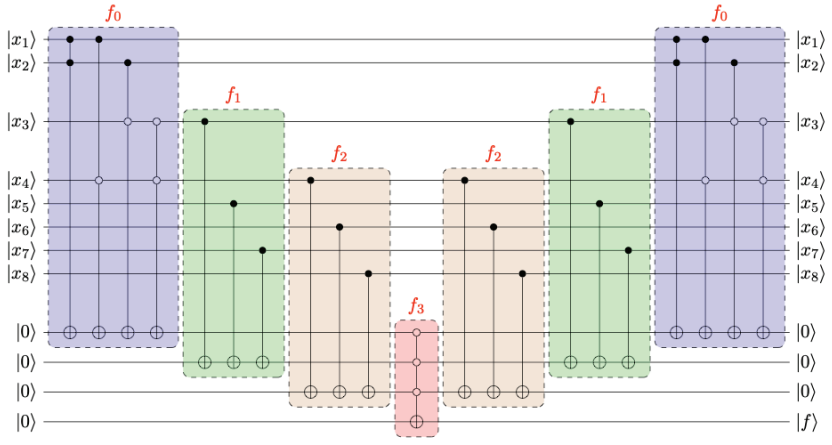


Note that the parameter k provides some control over the number of qubits the resulting circuit will have. Since each LUT is a truth table, the single-target gates in the above circuit are also defined using truth tables. Hence, the second compilation step is to translate these high-level operators into lower-level quantum ones.

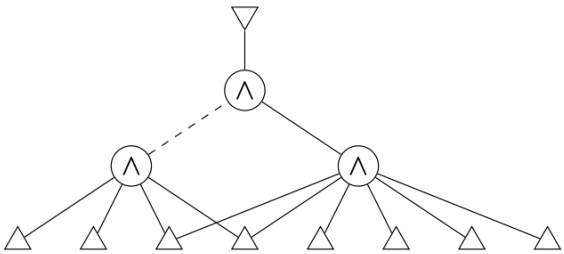
We can proceed in several ways. For example, we could synthesize the truth tables into Clifford+Rz circuits using spectral synthesis [135], or translate them into BDDs and use the technique introduced in Chapter 4 to extract ESOP expressions. In this case, we opt for directly deriving ESOPs from the truth tables using `pkrm_synth`:

$$
\begin{aligned}
f_0 &= x_0x_1 \oplus x_0\bar{x}_3 \oplus x_1\bar{x}_2 \oplus \bar{x}_2\bar{x}_3 \\
f_1 &= x_2 \oplus x_4 \oplus x_6 \\
f_2 &= x_3 \oplus x_5 \oplus x_7 \\
f_3 &= \bar{f}_0\bar{f}_1\bar{f}_2
\end{aligned}
$$

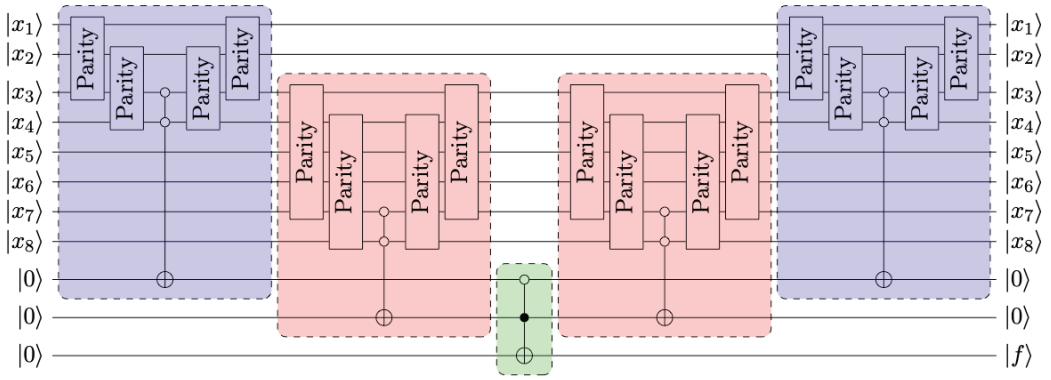
Now, we can straightforwardly map each expression to a sequence of multiple-control Toffoli gates, which, in turn, will be decomposed further into Clifford+T and so forth until we reach a representation based on the primitive operations provided by the underlying hardware.



XAG-based flow. The second flow is the method presented in Chapter 5. It starts by raising the level of abstraction. It transforms the original XAG into a high-level XAG, i.e., a XAG that implicitly represents most XOR gates as parity function, which are the inputs to the AND:



Next, it synthesizes a circuit based on parity and Toffoli instructions:



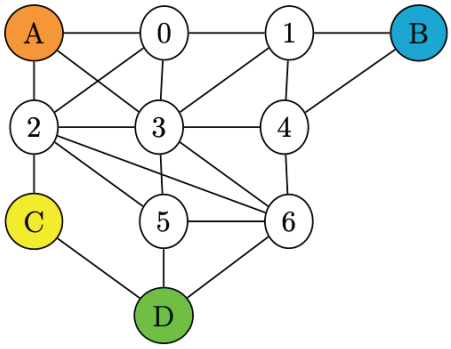
Then it lowers it to a circuit consisting of CNOT gates and Toffoli, which, in turn, is lowered to a circuit over the Clifford+T operator set.

Both flows lower the level of abstraction in small steps. This progressive lowering process enables the flows to discover more facts about the program, thus helping them find better optimization opportunities. For example, we can easily apply pebbling techniques to save qubits in a circuit defined using single-target gates because the information about the compute/cleanup pairs is readily available. Once this circuit is lowered into a sequence of Toffoli gates, we can more effectively apply cancellation. Indeed, identifying pairs of gates that cancel is generally easier when they are in their high-level form than in their Clifford+Tx implementation.

9.4.1 IBM's challenge: The Zed city problem

Zed city is a newly established (fictitious) municipality in Tokyo composed of 11 districts. Four convenience store chains A, B, C, and D have each built their first store in this new city. The goal is to use vertex coloring to distribute stores in the districts that still do not have one yet, while ensuring that there is only one store per district and that

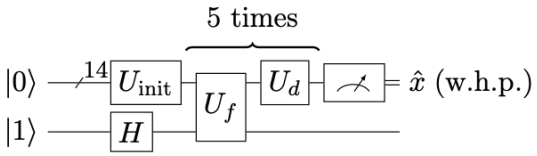
adjacent districts do not have stores from the same chain.



The Boolean modeling to solve this problem is similar to our previous example. However, in this challenge, we want to find a coloring using four colors: A ('00'), B ('01'), C ('10'), and D ('11'). Thus, we model the oracle's behavior using a function. Note that some vertices are already colored; we will assume that our initialization operator ensures not querying this function with inputs known beforehand to be non-solutions, i.e., an input that assigns color A to vertex v_0. The following Python function implements the desired behavior.

```
def f(v0, v1, v2, v3, v4, v5, v6 : BitVec(2)) -> BitVec(1):
    c1 = (v1[0] == v1[1]) and (v3 != v1)
    c023 = ((v0 ^ v2 ^ v3) == '00')
    c4 = (v4 != v1) and (v4 != v3)
    c5 = (v5 != v2) and (v5 != v3)
    c6 = ((v2 ^ v3 ^ v5 ^ v6) == '00') and (v6 != v4)
    return c1 and c023 and c4 and c5 and c6
```

IBM's challenge states that only five iterations are necessary to solve the problem when using a simulator. Hence, we use the following high-level circuit to solve the problem.

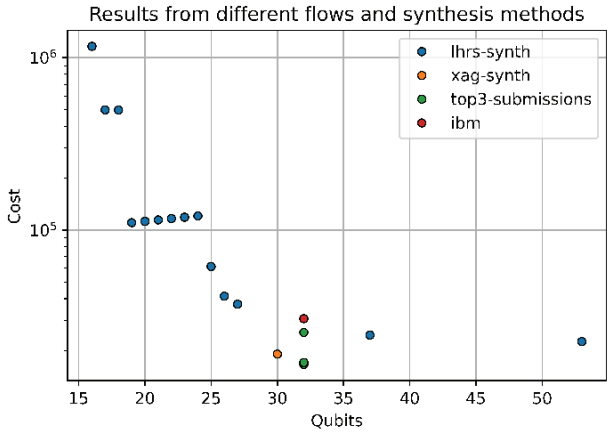


We evaluate our flow by solving the Zed city problem in IBM's challenge, which further imposes a constraint on the number of qubits: a solution must use at most 32 qubits. We use IBM's challenge as an example and evaluation because highly-optimized handcraft solutions are available [1]. We use these solutions as a baseline. First, we compare the code readability: Our Python implementation of f is objectively simpler to understand and implement than any of the submitted solutions—which define U_f in terms of low-level quantum operators.

In Figure 9.3, we report the results of compiling U_f using the flows described earlier. As baseline, we use IBM's sample solution and the top three submission with the same cost function as in the challenge, i.e., cost $= n_{1q} + 10 \cdot n_{2q}$, where n_{1q} is the number one-qubit operators and n_{2q} the number of two-qubit operators. The rationale behind

this cost assignment is that CNOT instructions have an error rate one order of magnitude larger than one-qubit instruction [2][1].

First, note that the XAG-based flow requires only 30 qubits while beating IBM's solution and coming reasonably close to the top two submissions in terms of cost. We observe a trade-off between the number of operations and qubits. Unfortunately, this flow could not use more qubits to minimize gates. By contrast, when using the k-LUT-based flow, we can adjust the values of k to explore the solution space.



	Qubits	cost
IBM's solution	32	30667
`Whit3z`	32	16613
`QunaVillage`	32	17053
`IIQ@QIC`	32	25490
XAG-based flow	30	19065
LHRS-based flows	16 - 53	22585 - 1163940

Figure 9.3: The results of implementing Grover's algorithm using different means. The cost function is cost $= n_{1q} + 10 \cdot n_{2q}$, where n_{1q} is the number one-qubit instructions and n_{2q} the number of two-qubit instructions.

9.5. SHOW CASE: MAPPING

Section 9.3.3 outlined two problems related to mapping quantum circuits to constrained devices: placement and routing. We briefly presented some of the techniques implemented in `tweeledum` to solve the mapping problem. This section will evaluate their effectiveness and performance by running benchmark circuits and comparing them against established compilers.

The frameworks used in this evaluation. We compare against two frameworks: IBM's `qiskit-terra 0.17.4` (Python) and CQC's `t|ket> 0.11.0` (C++). We empirically selected the combination of placement and routing techniques to obtain a good compromise between execution time and the quality of the solution:

- In `tweedledum`, we use `ApprxSatPlacer` and `LazyRouter`.

[1] At the time of the challenge. While this difference might have shrunk since then, it is still true that two-qubit operations are more prone to error than single-qubit ones [2].

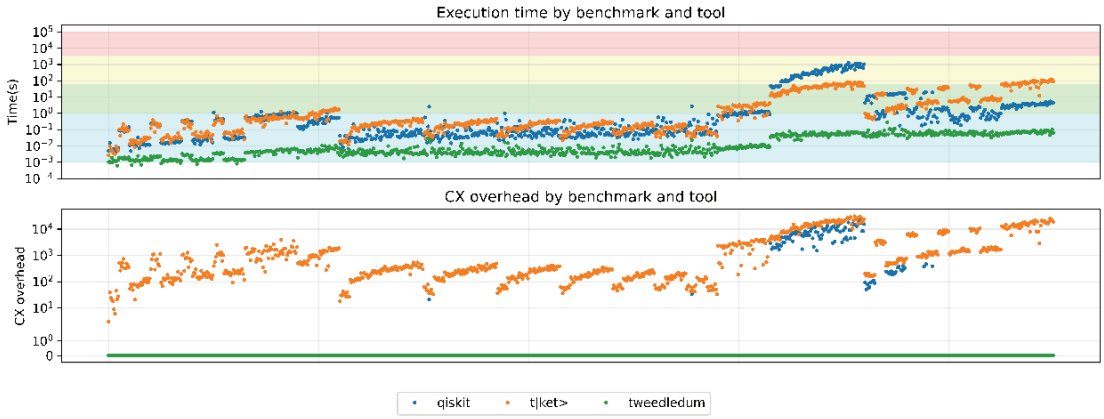


Figure 9.4: Comparison between mapping algorithms from `tweeledum`, `qiskit-terra`, and `t|ket>`, when using the QUEKO benchmark set.

- In `qiskit-terra`, we choose to use `CSPLayout` to try finding a perfect placement. If it fails, then we use a combination of SABRE layout and routing to do the mapping [86].
- In `t|ket>`, we disable the use of bridges and call its generic routing method, which we assume does the best job.

Runtime Environment. The tests were executed on a dedicated server featuring an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz with 256GB of RAM, running Linux Ubuntu 20.04.2 LTS (Focal Fossa). We installed all frameworks through `pip` and used Python version 3.8.5.

Benchmarks. We used the two sets of benchmarks to evaluate the mappers. The first set is the same used by Zulehner et al. [157] consists of 158 programs taken from "Reversible Logic Synthesis Benchmarks Page," [91] the RevLib collection [152], Quipper [68], and ScaffCC [74]. These benchmarks are used in several papers on mapping techniques [86, 48, 157]. The second set of benchmarks is the QUEKO benchmark suite [146].

Target architectures. The algorithms compared in this section work for any quantum architecture. For the first set of benchmarks, we use IBM Montreal architecture, which has 26 qubits. The QUEKO benchmarks define which of the following architectures should be used for each benchmark: Rigetti Aspen-4, IBM Tokyo, IBM Rochester, and Google Sycamore

Evaluation. Our evaluation of the results takes into consideration both efficiency and quality of the result. We evaluate the efficiency of each solution along one dimension: The time that each mapper needs to process a quantum circuit. We measure the quality of a solution by counting the number of controlled X instructions added by the mapping algorithm, the CX overhead.

Summary of Results. Figures 9.4 and 9.5 summarize the comparison between `tweeledum`, `qiskit-terra`, and `t|ket>`. Using the QUEKO benchmark set, the results

shown in Figure 9.4 demonstrate that `tweedledum` can perfectly map all benchmarks and be the fastest at doing it. We observe that `qiskit-terra` and `t|ket>` are, respectively, 42.28 and 83.05 times slower compared to our library on average. In our analysis of the raw data, we normalize the execution time of each benchmark using `tweedledum` as a baseline. Then we aggregate all the results using the geometric mean.

As shown in Figure 9.5, `tweeledum` does not perform as well on the other benchmark set. It is still significantly faster: On average, 276.6 times faster than `qiskit-terra` and 4.68 times faster than `t|ket>`. On the other hand, the quality of results is worse: 8% compared to `qiskit-terra` and 15% compared to `t|ket>`. (Note that our library also implements SABRE, which is even faster than the method used here while delivering the same quality of results as `qiskit-terra`. It does not, however, guarantee to find perfect placement when one exists.)

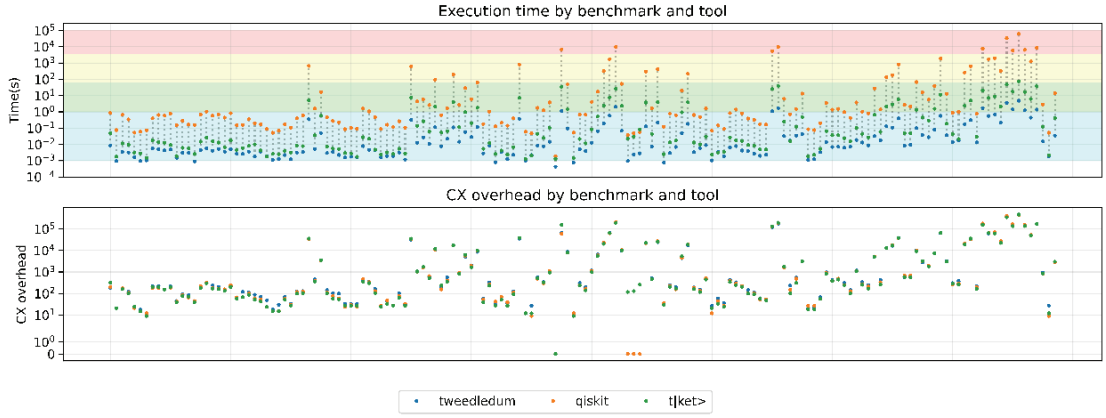


Figure 9.5: Comparison between mapping algorithms from `tweedledum`, `qiskit-terra`, and `t|ket>`, when using a benchmark set taken from various sources.

9.6. SUMMARY

It is widely believed that a language's expressive power influences the depth at which people can think. Existing quantum computing programming languages and frameworks limit the kinds of control structures, data structures, and abstractions developers can use; thus, the forms of algorithms they can construct are likewise limited. This chapter introduced `tweedledum`—a library that augments the expressive power of current frameworks by providing methods for synthesis, compilation, and optimization of quantum circuits. The library also seeks to create an open-source environment where state-of-the-art quantum compilation techniques can be developed and compared.

Finally, we see fast-paced changes in the field of quantum computing design tools. These changes are driven by various factors, ranging from enhanced compilation techniques and the desire to support higher levels of abstractions to technological advances in quantum hardware. While we cannot predict what the future will hold, the flexible design of `tweedledum` offers many possibilities for coping with future improvements.

Chapter 10

Conclusion

The research presented in this thesis focused on narrowing the gap between high-level quantum programs and technology-dependent implementations. We studied several techniques for synthesizing quantum operators that can be classically defined using Boolean functions. Various promising quantum algorithms rely on such operators, and, often, their cost dominates the total cost of a concrete implementation. On synthesis, we presented the following contributions:

- In Chapter 4, we presented an algorithm that can synthesize qubit-optimal circuits from large Boolean networks and a use-case that employs the technique to generate circuits using fewer T gates. We also mentioned how such a technique can be used in conjunction with hierarchical synthesis methods based on k-LUT networks to explore the trade-off between the number of qubits and the number of gates.
- Then we introduced a new flow for oracle synthesis that achieves better results than other state-of-the-art synthesizers in Chapter 5. We can reduce the number of qubits by 24.95% and the number of Clifford by 43.3% in the best cases. Crucially, these improvements were possible without increasing the number of T gates or the execution time.
- Finally, the last two chapters of Part II explored ways of synthesizing linear reversible circuits. We first focused on the family of permutation circuits, which appear in the context of quantum circuit mapping. These circuits can dominate the total costs for many applications. Chapter 6 described three algorithms to do the mapping itself or be used as a post-mapping optimization technique. All three algorithms guarantee local optimality, i.e., they generate permutation circuits with the optimal number of SWAPs or optimal depth. Chapter 7 proposed an SAT-based linear synthesis method that guarantees optimality when using CNOT gates. Although the latter is not scalable, we can use it with other heuristic synthesis methods to improve quality.

In Part III, we described the embodiment of our research contribution: a compiler companion library for the synthesis and compilation of quantum circuits called

`tweedledum`. The last chapter provided context on how our synthesis techniques fit quantum compilation's broader perspective and presented two use cases for our library and algorithms.

The presented work can empower the current frameworks and compilers to allow developers to use higher-level abstractions when implementing algorithms. The importance of this work for quantum computing is twofold:

- It is widely believed that a language's expressive power influences the depth at which people can think [126]. Existing quantum computing programming languages and frameworks limit the kinds of control structures, data structures, and abstractions developers can use; thus, the forms of algorithms they can construct are likewise limited. Therefore, the availability of more powerful compilers that can process more sophisticated, high-level languages can stimulate and aid the discovery of new quantum algorithms—which is crucial to quantum computing progress.

- Also, while existing NISQ devices can rely on a manual or semi-manual compilation process, which requires developers to describe algorithms in terms of basic unitary operators supported by the underlying machine, this will not be true for large-scale quantum computers. Larger devices will possess thousands of qubits and support error-correcting techniques. At such a scale, requiring developers to manually handle all the necessary complexity of implementing an algorithm will be impractical and likely to yield worse results.

Even now, the second point is essential. Albeit not having these large devices available, designing tools capable of handling large programs can play an indispensable role in resource estimation, i.e., estimating resources required to implement algorithms on future hardware. This allows us to understand better the power of quantum computing to make policy decisions, stimulate investment, and guide research towards technological advances that will lead to scalable devices.

10.1. FUTURE DIRECTIONS

This work is but a small step towards the strenuous goal of practical quantum computing. Many more will be needed before we get to the point of having reliable and scalable design tools empowering users with the ability to define quantum algorithms in a high level of abstraction while respecting the stringent constraints of real devices, let alone the necessary technology advances required for building programmable fault-tolerant machines. We close this thesis with some directions that might be taken for improving our current compilation stack.

Oracle synthesis. Several challenges remain, awaiting satisfactory solutions for the automated synthesis of large Boolean functions. Techniques that find a solution without exceeding a given number of ancillae are still rare. Indeed, our state-of-the-art techniques require reversible logic synthesis methods that need additional qubits. Typically, the

execution of the synthesis algorithm determines the number of ancilla qubits, i.e., it cannot be bounded a priori.

Also, in XAG-based synthesis, we greatly benefit from starting with an optimized network with lower multiplicative complexity. Thus, discovering new heuristics capable of further minimizing the number of AND gates would significantly impact the resources required to implement these networks as quantum circuits. Furthermore, the impact of the number of XOR nodes in the graph should be better studied. In our experiments, we have seen that the relation between the number of XOR steps and the number of CNOT gates is not straightforward.

The use of relative phase. Hierarchical synthesis based on k-LUT networks allows us to somewhat control the number of qubits in the resulting circuit through the selection of k. The price paid for such a rudimentary control is a more significant number of gates, which result from the synthesis of the incidental intermediate functions implemented by each LUT. As in XAG-based synthesis, we compute and clean up most of these functions, making them insensitive to phase errors. Therefore, further research into how to implement arbitrary relative phase functions might allow us to reduce the number of gates significantly, as it happens when implementing relative phase ANDs.

Tailored LUT mapping for quantum. Today we obtain a LUT network using classical logic synthesis algorithms that aim at minimizing the number of LUTs. However, when synthesizing quantum circuits, the number of LUTs might not be as influential as the functions these LUTs implement, i.e., we might be able to generate a less costly quantum circuit from a network with more LUTs if their functionality can more easily be implemented in quantum. The XAG-based synthesis technique indicates that to be the case: one can see a XAG as a LUT network in which the inputs to the AND gates are arbitrarily sized LUT implementing a parity function, and the AND gates are 2-LUTs.

Bibliography

[1] IBM Quantum Challenge. https://github.com/quantum-challenge/2019/tree/master/, 2019.

[2] IBM Quantum Experience. https://quantum-computing.ibm.com, 2019.

[3] IBM Unveils Breakthrough 127-Qubit Quantum Processor. https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor, November 2021.

[4] Scott Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, Cambridge, 2013.

[5] A. Abdollahi and M. Pedram. Analysis and Synthesis of Quantum Circuits by Using Quantum Decision Diagrams. In *Proceedings of the Design Automation & Test in Europe Conference*, pages 1–6, Munich, Germany, 2006. IEEE.

[6] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, Princeton University, NJ, Dept of Computer Science, 2012.

[7] Victor Arribas Abril, Pieter Maene, Nele Mertens, and NP Smart. Bristol fashion MPC circuits. https://homes.esat.kuleuven.be/{sim}nsmart/MPC/, 2019.

[8] Dorit Aharonov, Alexei Kitaev, and Noam Nisan. Quantum circuits with mixed states. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing - STOC '98*, pages 20–30, Dallas, Texas, United States, 1998. ACM Press.

[9] Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[10] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056, pages 430–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[11] Panos Aliferis, Daniel Gottesman, and John Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *arXiv:quant-ph/0504218*, October 2005.

[12] Noga Alon, F. R. K. Chung, and R. L. Graham. Routing Permutations on Graphs via Matchings. *SIAM Journal on Discrete Mathematics*, 7(3):513–530, May 1994.

[13] Noga Alon, Mauricio Karchmer, and Avi Wigderson. Linear Circuits over GF(2). *SIAM Journal on Computing*, 19(6):1064–1067, December 1990.

[14] Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS), 2015.

[15] Andris Ambainis. Understanding quantum algorithms via query complexity. In *Proceedings of the International Congress of Mathematicians (ICM 2018)*, pages 3265–3285, Rio de Janeiro, Brazil, May 2019. WORLD SCIENTIFIC.

[16] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, June 2013.

[17] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the controlled-NOT complexity of controlled-NOT–phase circuits. *Quantum Science and Technology*, 4(1):015002, 2018.

[18] Matthew Amy and Vlad Gheorghiu. Staq—A full-stack quantum processing toolkit. *Quantum Science and Technology*, 5(3):034016, 2020.

[19] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, October 2014.

[20] Ryan Babbush, Dominic W Berry, Ian D Kivlichan, Annie Y Wei, Peter J Love, and Alán Aspuru-Guzik. Exponentially more precise quantum simulation of fermions in second quantization. *New Journal of Physics*, 18(3):033032, March 2016.

[21] Aniruddha Bapat, Andrew M. Childs, Alexey V. Gorshkov, Samuel King, Eddie Schoute, and Hrishee Shastri. Quantum routing with fast reversals. *Quantum*, 5:533, August 2021.

[22] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, November 1995.

[23] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980.

[24] C. H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17(6):525–532, November 1973.

[25] Charles H Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[26] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. PennyLane: Automatic differentiation of hybrid quantum-classical computations. *arXiv:1811.04968 [physics, physics:quant-ph]*, February 2020.

[27] Ethan Bernstein and Umesh Vazirani. Quantum Complexity Theory. *SIAM Journal on Computing*, 26(5):1411–1473, October 1997.

[28] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, London UK, June 2020. ACM.

[29] Armin Biere, Marijn Heule, and Hans van Maaren, editors. *Handbook of Satisfiability*. Number volume 336 in Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam ; Washington, DC, second edition edition, 2021.

[30] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, Sept./1996.

[31] Joan Boyar and René Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Paola Festa, editors, *Experimental Algorithms*, volume 6049, pages 178–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[32] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight Bounds on Quantum Searching. *Fortschritte der Physik*, 46(4-5):493–505, June 1998.

[33] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz,

C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174, pages 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[34] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[35] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[36] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-Quantum Zero-Knowledge and Signatures from Symmetric-Key Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1825–1842, Dallas Texas USA, October 2017. ACM.

[37] Cynthia Chen, Bruno Schmitt, Helena Zhang, Lev S. Bishop, and Ali Javadi-Abhari. Optimizing quantum circuit synthesis for permutations using recursion. In *Proceedings of the 59th Annual Conference on Design Automation - DAC '22*, San Francisco, CA, USA, 2022. ACM Press.

[38] D. Chen and J. Cong. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 752–759, San Jose, CA, USA, 2004. IEEE.

[39] A. Chi-Chih Yao. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 352–361, Palo Alto, CA, USA, 1993. IEEE.

[40] Andrew M. Childs and Dominic W. Berry. Black-box Hamiltonian simulation and unitary implementation. *Quantum Information and Computation*, 12(1&2):pp0029–0062, January 2012.

[41] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, September 2017.

[42] Bob Coecke and Ross Duncan. Interacting Quantum Observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126, pages 298–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[43] Bob Coecke and Ross Duncan. Interacting quantum observables: Categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, April 2011.

[44] J. Cong and Yuzheng Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, Jan./1994.

[45] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC '71*, pages 151–158, Shaker Heights, Ohio, United States, 1971. ACM Press.

[46] Pedro C. S. Costa, Stephen Jordan, and Aaron Ostrander. Quantum Algorithm for Simulating the Wave Equation. *Physical Review A*, 99(1):012323, January 2019.

[47] Nicolas T. Courtois, Mourouzis Theodosis, and Daniel Hulme. Solving circuit optimisation problems in cryptography and cryptanalysis. *Cryptology ePrint Archive, Report 2011/475*, 2011.

[48] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. *arXiv:1902.08091 [quant-ph]*, page 32 pages, 2019.

[49] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, September 2019.

[50] Timothee Goubault de Brugiere, Marc Baboulin, Benoit Valiron, Simon Martiel, and Cyril Allouche. Reducing the Depth of Linear Reversible Quantum Circuits. *IEEE Transactions on Quantum Engineering*, 2:1–22, 2021.

[51] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, C. R. Ramakrishnan, and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[52] Alexis De Vos and Yvan Van Rentergem. Young subgroups for reversible computers. *Advances in Mathematics of Communications*, 2(2):183, 2008.

[53] David Deutsch. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, July 1985.

[54] David Elieser Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, September 1989.

[55] Cirq Developers. Cirq. Zenodo, August 2021.

[56] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 570–583, Valencia, Spain, May 2020. IEEE.

[57] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, July 1939.

[58] R. Drechsler. Pseudo-Kronecker expressions for symmetric functions. *IEEE Transactions on Computers*, 48(9):987–990, Sept./1999.

[59] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Enrico Giunchiglia, and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[60] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, June 1982.

[61] Magnus Gausdal Find. On the Complexity of Computing Two Nonlinearity Measures. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K. Vereshchagin, editors, *Computer Science - Theory and Applications*, volume 8476, pages 167–175. Springer International Publishing, Cham, 2014.

[62] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. Open source software in quantum computing. *PLOS ONE*, 13(12):e0208561, December 2018.

[63] Austin G. Fowler, Ashley M. Stephens, and Peter Groszkowski. High-threshold universal quantum computation on the surface code. *Physical Review A*, 80(5):052312, November 2009.

[64] Carsten Fuhs and Peter Schneider-Kamp. Synthesizing Shortest Linear Straight-Line Programs over GF(2) Using SAT. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, volume 6175, pages 71–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[65] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster Zero-Knowledge for boolean circuits. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1069–1083, Austin, TX, August 2016. USENIX Association.

[66] Craig Gidney. Halving the cost of quantum addition. *Quantum*, 2:74, June 2018.

[67] Dahmun Goudarzi and Matthieu Rivain. On the Multiplicative Complexity of Boolean Functions and Bitsliced Higher-Order Masking. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813, pages 457–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[68] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '13*, page 333, Seattle, Washington, USA, 2013. ACM Press.

[69] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing - STOC '96*, pages 212–219, Philadelphia, Pennsylvania, United States, 1996. ACM Press.

[70] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters*, 103(15):150502, October 2009.

[71] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[72] Wakaki Hattori and Shigeru Yamashita. Quantum Circuit Optimization by Changing the Gate Order for 2D Nearest Neighbor Architectures. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, volume 11106, pages 228–243. Springer International Publishing, Cham, 2018.

[73] Grace Murray Hopper. The education of a computer. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh) on - ACM '52*, pages 243–249, Pittsburgh, Pennsylvania, 1952. ACM Press.

[74] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[75] Cody Jones. Low-overhead constructions for the fault-tolerant Toffoli gate. *Physical Review A*, 87(2):022328, February 2013.

[76] Stephen Jordan. Quantum Algorithm Zoo. https://quantumalgorithmzoo.org, 2011.

[77] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.

[78] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry fields: A software platform for photonic quantum computing. *Quantum*, 3:129, 2019.

[79] Aleks Kissinger and Arianne Meijer-van de Griend. CNOT circuit extraction for topologically-constrained quantum memories. *arXiv:1904.00633 [quant-ph]*, May 2019.

[80] Donald E. Knuth. *Combinatorial Algorithms*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, Boston Columbus Indianapolis, 2011.

[81] Donald E. Knuth. *Satisfiablility*, volume 4, fascicle 6 of *The Art of Computer Programming*. Addison-Wesley, Boston Columbus Indianapolis, printing with corrections edition, 2018.

[82] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, December 2002.

[83] Samuel A. Kutin, David Petrie Moulton, and Lawren M. Smithline. Computation at a distance. *arXiv:quant-ph/0701194*, January 2007.

[84] Ryan LaRose. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum*, 3:130, March 2019.

[85] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, July 1959.

[86] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014, Providence RI USA, April 2019. ACM.

[87] Gai Liu and Zhiru Zhang. PIMap: A Flexible Framework for Improving LUT-Based Technology Mapping via Parallelized Iterative Optimization. *ACM Transactions on Reconfigurable Technology and Systems*, 11(4):1–23, December 2018.

[88] Guang Hao Low and Isaac L. Chuang. Hamiltonian Simulation by Qubitization. *Quantum*, 3:163, July 2019.

[89] Yuri Manin. *Vychislimoe i nevychislimoe [Computable and Noncomputable] (in Russian)*. Sovetskoye Radio, Moscow, 1980.

[90] D. Maslov, G. W. Dueck, and D. M. Miller. Techniques for the synthesis of reversible Toffoli networks. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):42, September 2007.

[91] Dmitri Maslov. Reversible logic synthesis benchmarks page. http://webhome.cs.uvic.ca/~dmaslov/, 2005.

[92] Dmitri Maslov. Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. *Physical Review A*, 93(2):022311, February 2016.

[93] Giulia Meuli, Bruno Schmitt, Rüdiger Ehlers, Heinz Riener, and Giovanni De Micheli. Evaluating ESOP Optimization Methods in Quantum Compilation Flows. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Reversible Computation*, volume 11497, pages 191–206. Springer International Publishing, Cham, 2019.

[94] Giulia Meuli, Mathias Soeken, Earl Campbell, Martin Roetteler, and Giovanni de Micheli. The Role of Multiplicative Complexity in Compiling Low T-count Oracle Circuits. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Westminster, CO, USA, November 2019. IEEE.

[95] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. SAT-based {CNOT, T} Quantum Circuit Synthesis. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, volume 11106, pages 175–188. Springer International Publishing, Cham, 2018.

[96] Giulia Meuli, Mathias Soeken, Martin Roetteler, Nikolaj Bjorner, and Giovanni De Micheli. Reversible Pebbling Game for Quantum Memory Management. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 288–291, Florence, Italy, March 2019. IEEE.

[97] D. M. Miller, D. Maslov, and G. W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 318–323, 2003.

[98] D. Michael Miller, Robert Wille, and Rolf Drechsler. Reducing Reversible Circuit Cost by Adding Lines. In *2010 40th IEEE International Symposium on Multiple-Valued Logic*, pages 217–222, Barcelona, Spain, 2010. IEEE.

[99] D.M. Miller and M.A. Thornton. QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, pages 30–30, Singapore, 2006. IEEE.

[100] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International on Design Automation Conference - DAC '93*, pages 272–277, Dallas, Texas, United States, 1993. ACM Press.

[101] Shin-ichi Minato. πDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695, pages 90–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[102] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd Annual Conference on Design Automation - DAC '06*, page 532, San Francisco, CA, USA, 2006. ACM Press.

[103] Alan Mishchenko and Marek Perkowski. Fast Heuristic Minimization of Exclusive-Sums-of-Products. In *Proceedings of RM'2001 Workshop*, pages 242–250, August 2001.

[104] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology*, 5(2):025010, March 2020.

[105] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010.

[106] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):86–99, January 2016.

[107] C. Paar. Optimized arithmetic for Reed-Solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*, page 250, Ulm, Germany, 1997. IEEE.

[108] Ketan N. Patel, Igor L. Markov, and John P. Hayes. Optimal Synthesis of Linear Reversible Circuits. *Quantum Info. Comput.*, 8(3):282–294, March 2008.

[109] Emil L. Post. *The Two-Valued Iterative Systems of Mathematical Logic. (AM-5)*. Princeton University Press, December 1942.

[110] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[111] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and Chao Chen. Mapping into LUT structures. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1579–1584, Dresden, March 2012. IEEE.

[112] M. Sadegh Riazi, Mojan Javaheripi, Siam U. Hussain, and Farinaz Koushanfar. MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 198–207, McLean, VA, USA, May 2019. IEEE.

[113] Heinz Riener, Rüdiger Ehlers, Bruno de O. Schmitt, and Giovanni De Micheli. Exact Synthesis of ESOP Forms. In Rolf Drechsler and Mathias Soeken, editors, *Advanced Boolean Techniques*, pages 177–194. Springer International Publishing, Cham, 2020.

[114] Vadim Ryvchin and Alexander Nadel. Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking. *Proceedings of SAT Competition 2018*, page 29, 2018.

[115] Mehdi Saeedi, Morteza Saheb Zamani, Mehdi Sedighi, and Zahra Sasanian. Reversible circuit synthesis using a cycle-based approach. *ACM Journal on Emerging Technologies in Computing Systems*, 6(4):1–26, December 2010.

[116] Sean Safarpour, Andreas Veneris, Gregg Baeckler, and Richard Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. In *Proceedings of the 43rd Annual Conference on Design Automation - DAC '06*, page 466, San Francisco, CA, USA, 2006. ACM Press.

[117] Tsutomu Sasao, editor. *Logic Synthesis and Optimization*, volume 212 of *The Kluwer International Series in Engineering and Computer Science*. Springer US, Boston, MA, 1993.

[118] Artur Scherer, Benoît Valiron, Siun-Chuon Mau, Scott Alexander, Eric van den Berg, and Thomas E. Chapuran. Concrete resource analysis of the quantum linear system algorithm used to compute the electromagnetic scattering cross section of a 2D target. *Quantum Information Processing*, 16(3):60, March 2017.

[119] Bruno Schmitt and Giovanni De Micheli. `tweedledum`: A Compiler Companion for Quantum Computing. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium, March 2022. IEEE.

[120] Bruno Schmitt, Ali Javadi-Abhari, and Giovanni De Micheli. Compilation flow for classically defined quantum operations. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 964–967, Grenoble, France, February 2021. IEEE.

[121] Bruno Schmitt, Alan Mishchenko, and Robert Brayton. SAT-based area recovery in structural technology mapping. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 586–591, Jeju, January 2018. IEEE.

[122] Bruno Schmitt, Fereshte Mozafari, Giulia Meuli, Heinz Riener, and Giovanni De Micheli. From Boolean functions to quantum circuits: A scalable quantum compilation flow in C++. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1044–1049, Grenoble, France, February 2021. IEEE.

[123] Bruno Schmitt, Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. Scaling-up ESOP Synthesis for Quantum Compilation. In *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 13–18, Fredericton, NB, Canada, May 2019. IEEE.

[124] Bruno Schmitt, Mathias Soeken, and Giovanni De Micheli. Symbolic Algorithms for Token Swapping. In *2020 IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 28–33, Miyazaki, Japan, November 2020. IEEE.

[125] Norbert Schuch and Jens Siewert. Programmable Networks for Quantum Algorithms. *Physical Review Letters*, 91(2):027902, July 2003.

[126] Robert W. Sebesta. *Concepts of Programming Languages.* Always Learning. Pearson, Boston Munich, eleventh edition, global edition edition, 2016.

[127] V.V. Shende, A.K. Prasad, I.L. Markov, and J.P. Hayes. Synthesis of reversible logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):710–722, June 2003.

[128] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Review*, 41(2):303–332, January 1999.

[129] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. T| ket>: A retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, 2020.

[130] Kaitlin Smith, Mathias Soeken, Bruno Schmitt, Giovanni De Micheli, and Mitchell Thornton. Using ZDDs in the Mapping of Quantum Circuits. 2019.

[131] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A Practical Quantum Instruction Set Architecture. *arXiv:1608.03355 [quant-ph]*, February 2017.

[132] Robert S Smith, Eric C Peterson, Mark G Skilbeck, and Erik J Davis. An open-source, industrial-strength optimizing compiler for quantum programs. *Quantum Science and Technology*, 5(4):044001, 2020.

[133] Mathias Soeken, Gerhard W. Dueck, and D. Michael Miller. A Fast Symbolic Transformation Based Algorithm for Reversible Logic Synthesis. In Simon Devitt and Ivan Lanese, editors, *Reversible Computation*, volume 9720, pages 307–321. Springer International Publishing, Cham, 2016.

[134] Mathias Soeken, Giulia Meuli, Bruno Schmitt, Fereshte Mozafari, Heinz Riener, and Giovanni De Micheli. Boolean satisfiability in quantum compilation. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2164):20190161, February 2020.

[135] Mathias Soeken, Fereshte Mozafari, Bruno Schmitt, and Giovanni De Micheli. Compiling Permutations for Superconducting QPUs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1349–1354, Florence, Italy, March 2019. IEEE.

[136] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. The EPFL logic synthesis libraries. *preprint arXiv:1805.05121*, 2018.

[137] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. Logic Synthesis for Quantum Computing. *arXiv:1706.02721 [quant-ph]*, June 2017.

[138] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. LUT-Based Hierarchical Reversible Logic Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(9):1675–1688, September 2019.

[139] Mathias Soeken, Laura Tague, Gerhard W. Dueck, and Rolf Drechsler. Ancilla-free synthesis of large reversible functions using binary decision diagrams. *Journal of Symbolic Computation*, 73:1–26, March 2016.

[140] Mathias Soeken, Robert Wille, Christoph Hilken, Nils Przigoda, and Rolf Drechsler. Synthesis of reversible circuits with minimal lines for large functions. In *17th Asia and South Pacific Design Automation Conference*, pages 85–92, Sydney, Australia, January 2012. IEEE.

[141] Fabio Somenzi. CUDD: Colorado university decision diagram package, 1996.

[142] Damian S Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: An open source software framework for quantum computing. *Quantum*, 2:49, 2018.

[143] Adrien Suau, Gabriel Staffelbach, and Henri Calandra. Practical Quantum Computing: Solving the Wave Equation Using a Quantum Approach. *ACM Transactions on Quantum Computing*, 2(1):1–35, April 2021.

[144] Pavel Surynek. Finding Optimal Solutions to Token Swapping by Conflict-Based Search and Reduction to SAT. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 592–599, Volos, Greece, November 2018. IEEE.

[145] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*, pages 1–10, Vienna, Austria, 2018. ACM Press.

[146] Bochen Tan and Jason Cong. Optimality Study of Existing Quantum Computing Layout Synthesis Tools. *IEEE Transactions on Computers*, pages 1–1, 2020.

[147] Eleonora Testa, Mathias Soeken, Luca Amarù, and Giovanni De Micheli. Reducing the multiplicative complexity in logic networks for cryptography and security applications. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[148] Eleonora Testa, Mathias Soeken, Heinz Riener, Luca Amaru, and Giovanni De Micheli. A Logic Synthesis Toolbox for Reducing the Multiplicative Complexity in Logic Networks. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 568–573, Grenoble, France, March 2020. IEEE.

[149] Tommaso Toffoli. Bicontinuous extensions of invertible combinatorial functions. *Mathematical Systems Theory*, 14(1):13–23, December 1981.

[150] Matthew Treinish, Jay Gambetta, Paul Nation, Paul Kassebaum, Qiskit-Bot, Diego M. Rodríguez, Salvador De La Puente González, Shaohan Hu, Kevin Krsulich, Laura Zdanski, Jessie Yu, Jim Garrison, Julien Gacon, David McKay, Juan Gomez, Lauren Capelluto, Travis-S-IBM, Manoel Marques, Ashish Panigrahi, Jake Lishman, Lerongil, Rafey Iqbal Rahman, Steve Wood, Luciano Bello, Divyanshu Singh, Drew, Eli Arbel, Joachim Schwarm, Jonathan Daniel, and MELVIN GEORGE. Qiskit/qiskit: Qiskit 0.34.2. Zenodo, February 2022.

[151] Meltem Turan Sönmez and René Peralta. The Multiplicative Complexity of Boolean Functions on Four and Five Variables. In Thomas Eisenbarth and Erdinç Öztürk, editors, *Lightweight Cryptography for Security and Privacy*, volume 8898, pages 21–33. Springer International Publishing, Cham, 2015.

[152] Robert Wille, Daniel Gro, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *38th International Symposium on Multiple Valued Logic (Ismvl 2008)*, pages 220–225, Dallas, TX, USA, May 2008. IEEE.

[153] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, October 1982.

[154] Bujiao Wu, Xiaoyu He, Shuai Yang, Lifu Shou, Guojing Tian, Jialin Zhang, and Xiaoming Sun. Optimization of CNOT circuits under topological constraints. *arXiv:1910.14478 [quant-ph]*, August 2021.

[155] Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theoretical Computer Science*, 586:81–94, June 2015.

[156] Louxin Zhang. Optimal Bounds for Matching Routing on Trees. *SIAM Journal on Discrete Mathematics*, 12(1):64–77, January 1999.

[157] Alwin Zulehner, Alexandru Paler, and Robert Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, July 2019.

Bruno **Schmitt**

PH.D CANDIDATE

Avenue de la Harpe 30, Lausanne, 1007, Switzerland

+41 797 43 24 99 | bruno.schmitt@epfl.ch | boschmitt | bo-schmitt

With more than 11 years of working experience in both academia and industry, I have strong software development skills, specifically in C/C++ as well as Python and basic Rust. As part of my research, I am the main developer and maintainer of an open-source full-stack library for quantum compilation, *tweedledum*, and active contributor for the EPFL logic synthesis libraries. My primary research interests include quantum computing, logic synthesis, formal verification, design automation tools (CAD) and SAT solvers.

Education

École Polytechnique Fédérale de Lausanne (EPFL) — *Lausanne, Switzerland*

PH.D IN COMPUTER SCIENCE — *Sep. 2017 - June. 2022 (Expected)*

- Title: Practical Compilation of Quantum Programs
- Advisor: Giovanni De Micheli
- Co-Advisor: Mathias Soeken

Universidade Federal do Rio Grande do Sul (UFRGS) — *Porto Alegre, Brazil*

B.S. IN COMPUTER ENGINEERING — *Aug. 2011 - Dec. 2016*

- Title: Fast Extract with Cube Hashing
- Advisor: André Inácio Reis
- Co-Advisor: Alan Mishchenko, UC Berkeley, CA, USA

ENSEIR-MATMECA, IPB — *Bordeaux, France*

1 YEAR EXCHANGE ELECTRICAL ENGINEERING — *Aug. 2013 - Jul. 2014*

- I was selected to participate in the BRAFITEC—sandwich degree program with French engineering schools.

Publications

Optimizing Quantum Circuit Synthesis for Permutations using Recursive Methods — *DAC'22*

C. CHEN, H. ZHANG, **B. SCHMITT**, L. S. BISHOP AND A. JAVADI-ABHARI — *San Francisco, United States*

Optimizing Quantum Circuit Synthesis for Permutations on Limited Connectivity Topologies — *APS March'22*

C. CHEN, H. ZHANG, **B. SCHMITT**, L. S. BISHOP AND A. JAVADI-ABHARI — *Chicago, United States*

`tweedledum`: A Compiler Companion for Quantum Computing — *DATE'22*

B. SCHMITT, G. DE MICHELI — *Antwerp, Belgium*

Compilation flow for classically defined quantum operations — *DATE'21*

B. SCHMITT, A. JAVADI-ABHARI, G. DE MICHELI — *Grenoble, France*

From Boolean functions to quantum circuits: A scalable quantum compilation flow in C++ — *DATE'21*

B. SCHMITT, F. MOZAFARI, G. MEULI, H. RIENER, G. DE MICHELI — *Grenoble, France*

Symbolic Algorithms for Token Swapping — *ISMVL'20*

B. SCHMITT, M. SOEKEN, G. DE MICHELI — *Miyazaki, Japan*

Evaluating ESOP Optimization Methods in Quantum Compilation Flows — *RC'19*

G. MEULI, **B. SCHMITT**, R. EHLERS, H. RIENER, G. DE MICHELI — *Lausanne, Switzerland*

Using ZDDs in the mapping of quantum circuits — *QPL'19*

K. SMITH, M. SOEKEN, **B. SCHMITT**, G. DE MICHELI — *Orange, United States*

Scaling-up ESOP Synthesis for Quantum Compilation — *ISMVL'19*
B. Schmitt, M. Soeken, A. Mishchenko, G. De Micheli — *Fredericton, Canada*

Compiling permutations for superconducting QPUs — *DATE'19*
M. Soeken, F. Mozafari, **B. Schmitt**, G. De Micheli — *Florence, Italy*

Exact Synthesis of ESOP Forms — *IWSBP'18*
H. Riener, R. Ehlers, **B. Schmitt**, G. De Micheli — *Bremen, Germany*

SAT-Based Area Recovery in Structural Technology Mapping — *ASP-DAC'18*
B. Schmitt, A. Mishchenko, R. Brayton — *Jeju Island, South Korea*

Fast Extract with Cube Hashing — *ASP-DAC'17*
B. Schmitt, A. Mishchenko, V. Kravets, R. Brayton, A. Reis — *Chiba, Japan*

Honors & Awards

2017 **IC School Ph.D Fellowship**, École Polytechnique Fédérale de Lausanne (EPFL)
2015 **A. Richard Newton Young Student Fellow**, Design Automation Conference (DAC)

Extra Scientific Work

Conference on Reversible Computation — *RC'22*
Program committee — *Urbino, Italy*

Architectural Support for Programming Languages and Operating Systems — *APLOS'22*
Reviewer — *Lausanne, Switzerland*

Transactions on Computer-Aided Design of Integrated Circuits and Systems — *TCAD'22*
Reviewer
- 2× in 2022.

Workshop on Open-Source EDA Technology — *WOSET'21*
Program committee — *Munich , Germany*

IEEE Internet Computing — *IC'21*
Reviewer

International Conference on Quantum Computing and Engineering — *QCE'21*
Reviewer

Journal on Emerging Technologies in Computing Systems — *JETC'20*
Reviewer

Transactions on Very Large Scale Integration Systems — *TVLSI'19*
Reviewer

International Conference on Tools with Artificial Intelligence — *ICTAI'19*
Reviewer — *Portland, United States*

International Symposium on Multiple-Valued Logic — *ISMVL'18*
Reviewer — *Linz, Austria*

Work Experience

Ph.D Research Assistant
Lausanne, Switzerland

École Polytechnique Fédérale de Lausanne (EPFL) — *Sep. 2017 - Present*

- Development and implementation of algorithms for quantum compilation.
- Maintainer of *tweedledum*—an open source library for writing, manipulating, and optimizing quantum circuits.

Research Intern - Quantum Computing
Redmond, United States

Microsoft — *June. - Sep. 2019 | July. - Oct. 2021*

- I have been twice to Microsoft as a research intern.

Research Intern - Quantum Computing
Zurich, Switzerland

IBM — *May. 2020 - Aug. 2020*

- I worked in the *qiskit-terra* core development team.
- Integrated *tweedledum* into *qiskit-terra*—adding oracle synthesis capabilities.
- Implemented many performance improvements across the stack.
- Collaborated closely with the research-only teams in implementing algorithms for synthesis and optimization of quantum circuits.
- Mentored other interns.

Visiting Researcher
Lausanne, Switzerland

École Polytechnique Fédérale de Lausanne (EPFL) — *May. 2017 - Sep. 2017*

- I worked on reversible logic synthesis and optimization.

Visiting Researcher
Berkeley, United States

University of California, Berkeley — *Jan. 2016 - May. 2017*

- I worked for the logic synthesis and verification group of Prof. Robert K. Brayton at UC Berkeley contributing to the development of ABC.
- Development and implementation of algorithms for logic synthesis and optimization.
- Improving ABC SAT solver.
- Implementation of a new SAT solver for ABC (satoko).
- Improving ABC area recovering heuristics during technology mapping (satlut).

Hardware Engineer Intern
Porto Alegre, Brazil

AEL Sistemas — *Jan. 2012 - Jun. 2013*

- Requirements capture based on client's specification.
- Collaborated with the specification of a FPGA design to be used in aerospace applications.
- Wrote documentation. (Design specification, implementation details and user manual).
- Implemented different parts of the design using VHDL.
- Conceived testbenches and test vectors for verification.
- Simulation.

Software Engineer Intern
Cachoeirinha, Brazil

Parks S/A Comunicações Digitais — *Mar. 2011 - Jul. 2011*

- Developed software for embedded systems used in telecommunication equipment.
- Developed drivers for devices implemented in FPGA.
- Conceived new functions and features to meet the customer's needs and specifications.
- Linux kernel development.

Extracurricular Activity

- Part of the organization committee of the Quantum Computing Hard- and Software Summer School (QCHS) 2021 at EFPL.
- Founding member and Vice-president (2019-2021) of EPFL Quantum Computing Association.
- Chair of IEEE Circuit and Systems Society (CASS) Student Branch (2012)
- High school exchange: 6 months, Bayfield HighSchool - Dunedin, New Zealand. (2006)
- Traveled as a backpacker in several countries. In 2008, a month in Western Europe. In 2011, a month in the United States. In 2014, one month in countries of Central and Eastern Europe followed by a month in Japan. In 2017 traveled one month in Canada.
- One of the selected Brazilian students to participate in the 2nd Advanced Microsystems Technologies for Sensor Applications Summer School. Summer school, winter for the southern hemisphere, sponsored by the German government (DAAD)